Generating Efficient Layouts from Optimized MOS Circuit Schematics

RLE Technical Report No. 535

January, 1988

Donald George Baltus

Research Laboratory of Electronics
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

This research was supported by the U.S. Air Force - Office of Scientific Research Grant AFOSR-86-0164, by a National Science Foundation Graduate Fellowship, and by Thinking Machines Corporation.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Research Laboratory of Electronics Massachusetts Institute of Technology / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) 77 Massachusetts Avenue Cambridge, MA 02139	7b ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Air Force Office of Scientific Research / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR 86-0164
8c. ADDRESS (City, State, and ZIP Code) Building 410 Bolling Air Force Base D.C. 20332 - 6448	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
	2305/B4		

11. TITLE (Include Security Classification)
The Design of High Performance Circuits for Digital Signal Processing

12. PERSONAL AUTHOR(S)
Jonathan Allen

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical Report 535	FROM ______ TO ______	January 1988	193

16. SUPPLEMENTARY NOTATION This Technical Report is based on the SM Thesis by Donald G. Baltus "Generating Efficient Layouts from Optimized MOS Circuit Schematics"

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

see next page

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Kyra M. Hall - RLE Contract Reports	22b. TELEPHONE (Include Area Code) (617) 253-2569 / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. ABSTRACT

A technique has been developed for efficiently mapping arbitrary MOS circuit schematics into corresponding layouts. Since transistor sizings are preserved during this transformation to layout, this synthesis methodology can effectively be applied to optimized circuit schematics which typically include such sized transistors. The technique is based on a restructuring of the circuit description hierarchy into a new hierarchy based on topological rather than functional constraints. The method of restructuring was selected so as to ease the mapping of the leaf subcircuits into layout as well as to ensure that a majority of the possible local layout optimizations (such as connection by abutment) could be effected at the leaf subcircuit level. This allows the subsequent placement and routing of the generated leaf sublayouts to be done using conventional placement and routing techniques without a substantial area or performance penalty over a true full custom design.

A program has been developed to implement this technique. Preliminary experiments in which the program has been applied to a number of module-size circuits have indicated that the layouts generated using this methodology are often superior in area and performance to those generated using more conventional automated layout techniques.

Table of Contents

List of Figures

List of Tables

Generating Efficient Layouts From Optimized MOS Circuit Schematics

by

Donald George Baltus

Submitted to the
Department of Electrical Engineering and Computer Science
on January 29, 1988.

Abstract

A technique has been developed for efficiently mapping arbitrary MOS circuit schematics into corresponding layouts. Since transistor sizings are preserved during this transformation to layout, this synthesis methodology can effectively be applied to optimized circuit schematics which typically include such sized transistors. The technique is based on a restructuring of the circuit description hierarchy into a new hierarchy based on topological rather than functional constraints. The method of restructuring was selected so as to ease the mapping of the leaf subcircuits into layout as well as to ensure that a majority of the possible local layout optimizations (such as connection by abutment) could be effected at the leaf subcircuit level. This allows the subsequent placement and routing of the generated leaf sublayouts to be done using conventional placement and routing techniques without a substantial area or performance penalty over a true full custom design.

A program has been developed to implement this technique. Preliminary experiments in which the program has been applied to a number of module-size circuits have indicated that the layouts generated using this methodology are often superior in area and performance to those generated using more conventional automated layout techniques.

Thesis Supervisor: Jonathan Allen

Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank my thesis advisor, Jonathan Allen, for his guidance and encouragement throughout the course of this research.

I am also grateful to Symbolics Inc. for providing a compactor which allowed my symbolic circuit descriptions to be mapped into layout. I would particularly like to thank Mark Matson for his help in the use of the compactor.

Finally, I would like to thank my colleagues at Thinking Machines Corporation for providing limitless encouragement and many insightful conversations during the summers I spent there at work on this research. I would especially like to thank Rolf Fiebrich for his invaluable guidance and unending support.

This research was supported by Air Force Office of Scientific Research Grant AFOSR-86-0164, by a National Science Foundation Graduate Fellowship, and by Thinking Machines Corporation.

Chapter 1
Introduction

In recent years, advances in integrated circuit technology have allowed circuit designers to implement systems of ever increasing size and complexity on a single chip. Integrated circuit design has thus become largely a problem of dealing with the complexity that is inherent in such large systems [1].

In order to simplify the task of implementing these complex integrated systems, structured design methodologies have been developed. These methodologies reduce the complexity of the integrated circuit design process while at the same time providing a standard framework to which automated design tools can be applied. Through the use of such automated tools, the circuit design process can be further simplified.

1.1. Structured Design

A number of structured design techniques have been developed for dealing with the problem of complexity in large software systems. Many of these same techniques can and have been applied to the similar problem of complexity in integrated circuit design. [2]

1.1.1. Abstraction

Perhaps the most important technique that can be used to reduce complexity in designing a large integrated system is the use of various levels of abstraction to represent and describe a design. In general, the same design can be described from a number of different views with each view corresponding to a specific aspect of the circuit design task. By specifying a design in some limited domain of representation, its description is often simplified sufficiently that many useful transformations and optimizations within that particular domain can be performed. More detailed descriptions of some typical representational views of a circuit design follow.

1.1.1.1. Functional Description

The functional description of a circuit design is simply a more formal version of the problem specification. The functional description of a circuit describes exactly how that design should behave in response to some set of inputs. A functional description can be as simple as a boolean expression which defines the behavior of a piece of combination logic or as complicated as a the specification of some detailed algorithm in a high level programming language. In some cases, delay specifications are also included in a functional description.

A functional specification should completely describe the desired behavior of a given circuit while at the same time providing no constraints on how the circuit which effects the desired behavior is to be implemented.

1.1.1.2. Structural Description

The structural specification of a circuit describes which components are used to realize a circuit's desired functionality as well as how those components are interconnected. A structural description may specify the particular interconnection of logic gates which is used to implement a given functional specification or similarly the particular interconnection of transistors that will implement some logic gate. Circuit level structural descriptions often include specifications of the sizes and types of transistors as well as of the sizes of node capacitances.

1.1.1.3. Physical Description

The physical description of a circuit defines exactly how a given circuit is to be implemented. At the highest level, a physical description may simply specify the relative placement of the functional blocks of a given design. At the lowest level, a physical description will specify completely the masks required for fabrication.

1.1.2. Hierarchy

Hierarchy is also used to reduce the complexity of a large design. The use of hierarchy involves subdividing a given design into some manageable number of submodules. This process of subdivision is then repeated in a continuing fashion on each of the submodules until the submodules are simple enough that they can be implemented in some straightforward manner. It is important to recognize that a given circuit design can be represented hierarchically in each of the different domains of representation described above.

The functional description of a circuit can be specified hierarchically with functions calling successively simpler functions until each procedure being called is simple enough that its behavior is well defined. Similarly, the physical structure of a circuit may be specified hierarchically with each level of physical module specifying only the placement and interconnect of its immediate submodules. It should be noted that while the hierarchies of the physical and functional descriptions of a given design are sometimes identical in structure, they can also be quite dissimilar. Often the hierarchy used for describing the functionality of a given design does not correspond well to the hierarchy that is appropriate for describing the physical structure of that same design. A typical example is that of a bit-slice design whose bit-slice physical hierarchy is different from the hierarchy best suited to describe its functionality.

1.1.3. Modularity

A modular design is one in which the interactions between different modules (as defined by the hierarchy) can be easily and well characterized. In the case of a functional module, this implies some known set of input and output signals through which any and all communication with other functional modules takes place. In the case of a physical module, it implies a known set of terminals through which all external connections are made. Modularity in a design allows the various constituent submodules to in large part be designed and tested before the system as a whole is composed.

1.2. Automatic Layout Tools

By applying the various techniques of structured design described in the previous section, the process of integrated circuit design can be reduced to a series of transformations from one level of representation to the next. A design is first specified in terms of its desired functionality. Next, a structural specification is generated from the functional description. Finally, a mask layout is generated from the structural description.

This ability to describe the circuit design process as a sequence of well defined transformations has allowed designers to develop tools which automate (either entirely or in part) the process of mapping a circuit description from one level of representation to the next.

Several of the most commonly used automated layout methodologies are described below.

1.2.1. Weinberger Arrays

Through use of a *Weinberger Array* any combinational logic function can be mapped into a regular array layout structure. Each gate in the logic network takes up one vertical strip in the array. The inputs and outputs to the array are generally restricted to appear at the left and right edges of the array. The width of the array is determined by the number of logic gates and the height of the array is determined by the number of interconnection tracks required. The height of the array can be minimized by selecting an ordering of the gates which minimizes the number of interconnection tracks required. [3]

1.2.2. PLA's

A *programmable logic array* is similar to a Weinberger array in that it allows arbitrary combinational functions to be realized using a regular array layout structure. Unlike the Weinberger Array however, a PLA utilizes two array planes (an AND plane and an OR plane) to realize the functionality of arbitrary sums of products boolean expressions. By jointly minimizing the various output functions the size of the PLA can be minimized. [4]

1.2.3. Gate Arrays

Whereas Weinberger arrays and PLA's are generally used to realize the functionality of only some part of a larger design, *gate arrays* are typically used to generate the layout of an entire chip. In a gate array design the locations of the transistors on the chip are fixed at specific sites arranged in an array configuration. Space is left between the transistors to allow subsequent personalizing routing to be done for individual designs. The input to a gate array system is generally a list of logic gates and their interconnections. A placement program maps these logic gates into sets of contiguous transistor sites on the chip. A routing program then provides the appropriate interconnections between the gates. The fixed transistor topology of gate array designs allows wafers to be fabricated through all steps short of metalization before any personalization is added. This reduces the mask costs of a given design while also greatly reducing the time from design to completed chip. [2]

1.2.4. Standard Cells

As with gate arrays, *standard cell* design systems are generally used to generate layouts of entire chips. The mapping from functionality to mask description is also similar to that of a gate array system. In the case of standard cells however, the exact locations of transistors are not fixed. Instead a library of fixed layouts corresponding to the various logic gates is used but these layouts may be placed anywhere on the chip. The standard cells are generally placed in rows with routing channels between the rows to provide space for the required interconnections.

1.2.5. Silicon Compilers

A number of more sophisticated so called *silicon compilers* have been developed for mapping more complex functional specifications into mask level descriptions. These systems generally utilize variations on a generic floor plan and architecture to realize some desired functionality. A typical example of this is the MacPitts Silicon Compiler [5] which converts functional specifications written in LISP into mask layout using a predetermined general architecture and floor plan.

1.3. Limitations of Conventional Automatic Layout Tools

Although the automated layout tools described in the previous section do allow appropriate mask level descriptions to be rapidly generated from functional specifications, the layouts generated by these synthesis tools generally sacrifice much in the way of area and performance as compared to layouts generated using a full custom design methodology. This section proposes a number of fundamental reasons for the inability of these conventional automated layout methodologies to generate high performance integrated circuit layouts.

1.3.1. Limited Exploration of Design Space

The usefulness of an automated layout system is directly related to the number of layout problems to which it can be successfully applied. As a result, automatic layout systems have been designed to be as general as possible. The generality of these tools however has for the most part been limited to the functional domain. While these systems generally allow a great deal of flexibility in the functionality of the circuits they can be used to realize, they at the same time focus on a quite limited set of circuit and layout methodologies to implement these designs.

The canonical silicon compiler system takes as input the functional specification of a design and generates as output a corresponding physical description. The preceding discussion of structured design methodologies however makes it clear that this mapping from the functional domain into the physical domain is not a one step transformation. An intermediate structural representation which specifies the exact circuit topology used to realize the desired functionality must also be generated.

This two step transformation is illustrated in figure 1-1.

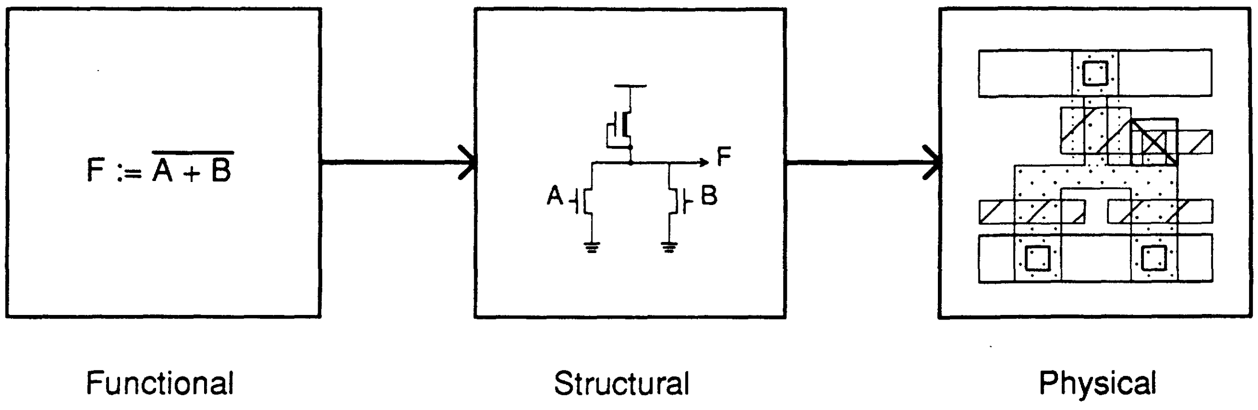

Figure 1-1: Functional to Physical Transformation

The two transformations involved in going from a functional to a physical specification are by no means equivalent in complexity or in difficulty of implementation. The transformation from a functional representation into a structural representation is simply a mapping from one abstract model into another. The transformation from a structural specification into a physical specification on the other hand is not so straightforward. This transformation must necessarily deal with the multitude of constraints that are involved in implementing an MOS circuit in a limited number of layers on a two-dimensional chip.

It is thus not surprising that most automatic layout systems synthesize designs by using only a very limited set of circuit and layout methodologies. Such systems generally utilize circuit methodologies for which well defined mappings into the physical domain are already known. The circuits used in PLA's or Weinberger Arrays are good typical examples.

The simplicity of the transformation from the functional to the structural domain allows these systems to easily coerce a given functional specification into the desired circuit methodology. The subsequent mapping into the physical domain is also straightforward since the circuit methodology used has been selected specifically to ease this transformation.

While this strategy for layout synthesis has been successfully employed by many systems which generate layouts from functional descriptions, the simplifying constraints inherent in such approaches introduce significant costs in the way of layout area and electrical performance. The restrictions on allowable circuit methodologies often prevent these systems from using the circuit topologies and transistor sizings best suited to particular design situations. In more abstract terms, it should be clear that an automated layout system which severely limits the useable topological and physical design spaces will rarely generate layouts which are comparable in performance to those generated using a more custom design methodology which allows the entire topological and physical design spaces to be explored.

1.3.2. Inappropriate Physical Hierarchy

Another limitation of conventional layout tools relates to the inability of these tools to generate final design description hierarchies which are appropriate for the physical domain of representation.

It is useful at this point to consider what is meant by an *appropriate* design description hierarchy. The hierarchy used in describing a system defines how that system is divided into submodules. If the hierarchy is such that the interactions between the various submodules can be easily and well characterized, then the design possesses the desirable attribute of *modularity* and the hierarchy which defines that design is in some sense appropriate. The hierarchy that is best suited to one representational view of given design may not be at all appropriate for describing a different view of that same design however. The same design hierarchy which subdivides a system in such a way as to generate a very modular functional description, may divide the system in a way that is very poorly suited to the physical domain.

The input to each of these synthesis tools is a functional design specification. The hierarchy that is initially present is thus not surprisingly more well suited to the functional domain than it is to the physical domain.

At the same time, the methods typically employed by these systems in transforming a hierarchical design description from one representational view into another do not involve any restructuring of the hierarchy present in the original view. The transformation of such a design representation is instead implemented as a set of simpler transformations on the leaf submodules of that representation.

Conventional automated layout methodologies thus tend to generate layout descriptions whose hierarchical structures are based on functional rather than physical design considerations. This can lead to a far from optimal physical hierarchy since many of the issues involved in defining an appropriate physical hierarchy are not apparent in the functional domain.

1.4. An Alternate Approach

This project investigates an alternate approach to layout synthesis through which many of the performance penalties associated with conventional automated layout systems can be avoided.

The preceding discussion makes it clear that a high performance layout synthesis system must provide the same flexibility in the area of structural to physical transformations as is already available in the realm of functional to structural mapping.

This project thus centers on developing a system for generating efficient layouts from arbitrary structural representations. This system for mapping arbitrary MOS circuit schematics into layout is intended for use as one component of a larger silicon compiler system. Such a compiler would be free to select circuit topologies and transistor sizings appropriate to specific design situations. The layouts generated by such a system would thus be expected to be of better quality than those generated by conventional synthesis systems which are by design constrained to utilize much smaller topological design spaces.

The synthesis methodology developed in this project also addresses the problem of inappropriate physical hierarchy. Before mapping a given MOS circuit schematic into layout, a new hierarchy of the design description is generated based on topological rather than functional constraints. The method of restructuring the circuit description hierarchy was selected so as to ease the mapping of the leaf subcircuits into layout as well as to ensure that a majority of the possible local layout optimizations (such as connection by abutment) could be effected at the leaf subcircuit level.

1.5. The Goals of the Research

In the area of integrated circuit design there is currently a tradeoff between the fast design times afforded by automated layout systems and the high performance generally associated with full custom designs.

It was hoped that by approaching the problem of automated layout using the novel methodology presented in this report many of the performance penalties generally associated with automatic layout systems could be avoided.

The primary goal of this body of research was thus to develop a program for rapidly generating *efficient* mask-level layouts from arbitrary MOS circuit schematics. In order to achieve this final goal a number of intermediate goals were also pursued.

It was clear that a system for mapping arbitrary MOS schematics into layout would require a layout methodology that could effectively be applied to many different classes of MOS circuits. Thus, one goal of this research was simply to investigate, develop and characterize efficient layout methodologies applicable to wide classes of MOS circuits.

It was also necessary to investigate the nature of design description hierarchies appropriate for the physical domain. Given such a characterization, it was hoped that an algorithmically based methodology for restructuring a circuit description hierarchy into a new description hierarchy more suited to the physical domain could be developed.

Finally, it was hoped that the usefulness of this restructuring could be demonstrated by showing that if the physical hierarchy of a design is properly structured then the placement and routing of the leaf modules of that hierarchy can be effected using conventional placement and routing techniques without a substantial area or performance penalty over a full custom design.

1.6. An Outline of the Report

This report consists of twelve chapters including this introduction. Chapter two provides a summary of existing work in the area of layout synthesis from circuit schematics. The third chapter discusses the basic design considerations which guided the development of the program. Chapter four describes the specification language used to provide input to the program. The next two chapters discuss the layout methodology employed by the program in realizing circuits as well as the exact method used to restructure the circuit description hierarchy for layout. Chapter seven and eight provide details of the specific algorithms used to implement the placement and routing phases of the program. In chapter nine the program's layout optimization procedures are described. Chapter ten illustrates the workings of the program through a detailed example. The next chapter presents a number of example layouts generated by the program and analyzes the results in terms of the goals of the research. The final chapter summarizes the research and discusses future improvements and directions.

Chapter 2
Schematic to Layout Systems

In this chapter a number of existing schematic to layout systems are described. Each of these systems generates the layout of a leaf cell given a circuit schematic of that cell as well as a set of appropriate boundary constraints. The methodologies used by these systems are summarized and finally a comparison and analysis of of the different approaches is made.

2.1. DUMBO

DUMBO is a system for compiling NMOS circuit descriptions into layout. [6] The input to DUMBO is a connectivity list of the circuit along with a specification of its external connections.

The program performs an initial placement of the circuit components using a force-directed placement methodology. Several subsequent phases serve to finalize the placement and complete the routing between components.

DUMBO is implemented as a pipeline of five separate programs. Each of these programs performs one of the compilation phases mentioned above. These five programs are listed (in the order they are called) and described below.

Placer	This program determines the relative positions of the transistors, gates, and external connections.
Orienter	This gives the placed transistors and gates their proper orientation and/or mirroring.
Expander	This program expands functional blocks such as inverters or logic gates into the appropriate transistor circuits.
Wirer	This generates the appropriate pairwise connections (branches) between nodes from the original net input.
Router	This program generates the final routing of each branch.

Each of these programs is implemented using standard algorithms similar to those used in more specific programs dedicated to the single task being performed.

The output produced by DUMBO is a symbolic layout which is subsequently compacted to generate a final mask level layout.

2.2. Koss's Topology Generator

A similar program for automatically generating layouts from NMOS circuit schematics was developed by Michael Koss at MIT. [7]

Koss's program uses a MINCUT algorithm to establish the relative placement of the transistors in the circuit. His methodology is unique in that it treats transistors and wires with equal weight during the placement process.

The MINCUT algorithm establishes the relative placement of the transistors as well as the relative locations of the connections between transistors. The locations of these connections are used to establish the proper rotation and orientation of each transistor.

Finally, a heuristic procedure is used to assign layers to the wire segments connecting the various transistors as well as external terminals.

As with DUMBO, Koss's program generates symbolic layout as its final output.

2.3. TALIB

TALIB is a knowledge-based system for automatically synthesizing leaf cell layouts from NMOS transistor and gate-level descriptions. [8]

The input to TALIB describes the circuit components (either transistors or logical gates) and their interconnections as well as any geometric constraints for the boundary of the generated layout. The geometric constraints specify which side of the layout a signal node is to appear on as well as optionally the exact ordering of the signals on a given side. These constraints can also specify physical constraints on the cell layout such as aspect ratio or bounds on a given dimension.

TALIB is implemented as a rule-based planning system. TALIB generates a final layout through the use of domain specific knowledge encoded as rules.

Most of the rules are simply subtasks which correspond roughly to a designers steps in the integrated circuit design process. These subtasks are called in a hierarchical manner with the more abstract subtasks calling several lower-level subtasks.

The remainder of TALIB's rules are *demons* which guide the program in the selection of subtasks.

The system's operation can be described as follows. TALIB continually uses the presently known facts to develop a plan structure. Once this structure is developed at a given level, each of the plan steps is expanded by calling the appropriate lower-level subtasks. This process continues until a solution is found or a constraint violation is detected. If a violation is detected, the system backtracks to undo some previous design decision and try a different approach.

Unlike the other systems described here, the output of TALIB is a mask level description of the completed layout of the cell.

2.4. Sc2

Sc2 is system for automatically generating CMOS layouts from transistor schematics or more high level descriptions. [9] The program can be applied to arbitrary CMOS circuits.

A design can enter Sc2 in one of three different ways. It can be described in terms of connected logic gates; it can be described by a circuit schematic; or it can be described in terms of boolean expressions. As with the other cell generation programs, constraints on the locations of external connections to the generated layout can also be specified.

If the input is not in the form of a circuit schematic, a *translation* program is invoked which produces a list of transistors and their sizes. This list of transistors is then fed into the layout portion of Sc2.

The design is next split into rows of logic. Each row of logic is then processed separately.

The target layout image for each row is a gate matrix consisting of a single row of n-MOS and a single row of p-MOS transistors. The placement and routing of each row of logic involves several steps.

1. Wide transistors are split into smaller transistors to be connected in parallel.
2. N and P devices sharing common gates are paired.
3. The pairs of devices are ordered within the row by use of the Kernigan-Lin algorithm.
4. Transistor pairs are oriented so as to maximize the number of adjacent sources and drains.
5. The program next connects the nodes of each net in the circuit while trying to minimize the amount of area used. The routing is performed using a number of different interconnection methods. All possible connections are made using the least expensive method and then the next method is tried. The methods of interconnect used (in order) are: abutting transistors, routing over transistors, routing over existing gates, and channel routing.
6. Finally, heuristics are used to eliminate unnecessary wire level changes.

The layout generated by Sc2 is a symbolic layout described in the "i" language.

2.5. The IBM Program

A program similar to Sc2 has been developed at IBM. [10] [11] This program also maps arbitrary CMOS circuit schematics into layout, and the target layout image is once again a gate matrix consisting of two parallel rows of n-MOS and p-MOS transistors.

This work differs from Sc2 in the way in which the positions and orientations of transistors in the gate matrix are selected. The program uses efficient search techniques and dynamic programming algorithms to select truly optimal transistor placement configurations. The placement and flipping heuristics used by Sc2 do not guarantee an optimal solution.

The transistor positioning and orientation procedure is outlined below.

1. A depth-first-search routine is used to expand potential transistor placements along the gate matrix array. A virtual search tree is generated on which a branch-and-bound procedure is performed. Only placements that can be subsequently routed are generated.
2. During the depth-first expansion, the optimal orientations of the transistors in each potential placement are determined.
3. The detailed routing of the best placement found is then performed.

The output generated by the program is symbolic layout.

2.6. TOPOLOGIZER

TOPOLOGIZER is an expert system for the automatic layout of CMOS cells from an input schematic description [12].

TOPOLOGIZER takes as input a transistor connectivity description as well as information about the environment in which the cell is to reside. The environment specification includes information about the aspect ratio of the cell as well as external connection constraints. The aspect ratio is specified in terms of transistors (width in columns and height in rows).

A symbolic layout is created by separately performing transistor placement and routing.

The aspect ratio constraint defines a grid on which the various transistors must be placed. Given such a placement grid, the placement problem is translated into the problem of finding a placement on this grid such that the number of nearby single-layer routing connections is maximized. Kollaritsch likens this problem to the game of dominoes. The placement expert first generates an initial random placement given the aspect ratio information. The expert then responds to recognized patterns to bring the placement incrementally closer to a final solution.

After the placement phase is completed, a routing expert is used to create a rough routed layout. The initial routing is functionally correct, but is very inefficient in the way of area and performance. The expert router thus continues by applying rules aimed at improving the routing.

The final output of TOPOLOGIZER is a symbolic layout compatible with the MULGA symbolic layout system.

2.7. Comparative Analysis

In this section a comparative analysis of the various synthesis approaches described here is presented. Table 2-1 lists each of the schematic to layout programs presented in this chapter and summarizes the attributes of each.

Summary of Schematic to Layout Programs					
Program	Technology	Programming Methodology	Layout Methodology	Layout Quality	Execution Time
DUMBO	NMOS	Algorithmic	General	Good	Very Poor
Koss's Program	NMOS	Algorithmic	General	Poor	Very Poor
TALIB	NMOS	Knowledge Based	General	Very Good	Fair
Sc2	CMOS	Algorithmic	Restricted	Fair	Very Good
IBM Program	CMOS	Algorithmic	Restricted	Fair	Very Good
TOPOLOGIZER	CMOS	Knowledge Based	Restricted	Good	Poor

Table 2-1: Summary of Schematic to Layout Programs

2.7.1. Generality

DUMBO, Koss's program and TALIB can only be used to realize NMOS circuits while Sc2, the IBM program and TOPOLOGIZER are applicable only to CMOS circuits. It is interesting to note the correlation between the layout methodologies employed by these systems and the circuit technologies to which they can be applied.

All the programs which synthesize NMOS circuits use general layout methodologies which can be applied to many different classes of circuits. These general layout methodologies can be employed only because the problems of well and substrate contact placement associated with CMOS circuit layout are avoided. The programs which can be applied to CMOS circuits are forced to use more restrictive layout methodologies which limit their usefulness and general applicability. While Sc2 and the IBM program can be applied to to arbitrary CMOS circuit topologies, the gate matrix target layout methodologies employed by these programs limit their *efficient* use to circuits which include equal numbers of n-MOS and p-MOS transistors. TOPOLOGIZER, on the other hand, uses a placement and layout technique which assumes that transistors are of similar size. The program thus has difficulty dealing with sized transistor circuits.

2.7.2. Computational Efficiency

TALIB and TOPOLOGIZER both use rule based systems to address the problem of layout synthesis. While there are many advantages associated with the ability to encode knowledge about the layout process as rules, the use of a knowledge based approach can introduce significant costs in the way of execution speed. Both TALIB and TOPOLOGIZER are computationally rather slow.

Although DUMBO, Koss's program, Sc2, and the IBM program all use algorithmically based programming methodologies, only Sc2 and the IBM work have any speed advantage over the programs which use a knowledge based approach. DUMBO and Koss's program are both relatively slow (taking in excess of twenty CPU minutes for circuits with on the order of twenty-five transistors). These programs have poor computational performance because they use general layout methodologies while making little use of the hierarchy present in VLSI designs. Neither introduces any hierarchy which is not included in the original input description. Problems thus occur when these two programs are applied to large circuits.

While Sc2 and the IBM program make little use of hierarchy, the computational advantage associated with the use of such restrictive layout methodologies serves to give Sc2 and the IBM program by far the fastest execution times of the programs presented here.

No specific execution time figures are given for TALIB, but it seems from the information available that TALIB is at least somewhat faster than DUMBO for small designs and that TALIB's performance advantage increases for larger designs. TALIB's ability to plan at various levels of abstraction allows it to make good use of the hierarchy present in a given design. This use of hierarchy allows TALIB to deal reasonably with large designs and is also a factor in TALIB's speed advantage over TOPOLOGIZER, the other knowledge based system presented here. TOPOLOGIZER applies its rules to a flattened circuit description and makes no use of hierarchy in generating a final layout.

2.7.3. Layout Efficiency

Of the programs presented here, DUMBO and TALIB generate the layouts of the highest quality in the way of area and electrical performance. The general layout methodologies employed by these programs allow many different classes of circuits to be realized in an efficient manner. Although Koss's program also uses a general layout methodology, the layouts generated by that program are of noticeably lower quality. Koss's program works with a flattened circuit description and allows no grouping of transistors during the layout process. Individual transistors are thus the largest objects operated on during the placement phase of the program. This prevents the program from performing the local layout optimizations that are essential in the generation of efficient layouts.

One additional factor that contributes to the high quality of the layouts generated by TALIB involves the use of boundary constraints. It is possible for a user to guide TALIB's layout process through the liberal use of boundary constraints in the input specification. The boundary constraints used by TALIB include position constraints on power busses as well as on signal lines. By selecting the relative positions of multiple power busses as well as signal lines, a user can guide TALIB towards some desired layout. While it may be desirable to allow a user to aid the tool in this way, it is important to realize that the layouts generated by TALIB would be of a lesser quality if the tool were used in an environment where the boundary constraints were generated automatically rather than by a human designer.

The programs applicable to CMOS circuits use restrictive layout methodologies and thus these programs can generate inefficient layouts when applied to non-standard circuits. Of the CMOS programs, TOPOLOGIZER generates the best layouts when applied to circuits of arbitrary structure. Because Sc2 and the IBM program employ more restrictive gate matrix layout methodologies, each pays a high performance penalty when applied to circuits including unequal numbers of n-MOS and p-MOS transistors. This sacrifice of layout quality does allow them to run very quickly however.

Chapter 3
Design Considerations

In this chapter the high level design considerations involved in the development of the synthesis program, SOLO[1], are discussed and a rough outline of the program's structure is presented.

3.1. System Requirements

The primary goal of this research was to develop a program for mapping arbitrary MOS circuit schematics into efficient layouts. The first step in the design process was to translate this general goal into a more concrete set of specifications.

3.1.1. Generality

While it is desirable for an automated layout system to be as general in its applicability as possible, each added degree of flexibility introduces new costs in the way of layout quality and computational efficiency. A tool designed specifically for a small subset of problems will generally produce better solutions than a tool which must be more generally applicable. This section outlines the factors that were considered in determining the minimal level of generality SOLO should maintain in a number of different areas.

3.1.1.1. Circuit Topologies

It was essential that SOLO be flexible in the types of circuits it could be used to realize. One of the fundamental advantages of a system which generates layouts from schematics rather than from functional specifications is the ability of such a system to use the circuit topologies and transistor sizings best suited to particular design situations. Much of this advantage is lost if severe restrictions are placed on the form of

[1]SOLO stands for Schematic tO LayOut program.

the input circuits. A system which could be used to generate layouts from arbitrary MOS circuit topologies thus became the target.

3.1.1.2. Technology

It was clear that the program should be as technology independent as possible. The design time associated with developing a system of this size is such that it would be inefficient to develop a system which would become obsolete as soon as small changes in the available technology occur. SOLO was as a result structured to be easily adaptable to changes in design rules.

It was also necessary to consider whether the system should be applicable only to NMOS circuits, only to CMOS circuits, or to both NMOS and CMOS circuits. Since it was anticipated that the system would employ a general layout methodology, it seemed that it would be best suited to the realization of NMOS circuit designs. Recall that general layout methodologies were successfully employed by a number of schematic to layout systems in realizing NMOS circuits. At the same time however, the increasing dominance of CMOS technologies made it clear that in order for the program to be of general use now and of any use in the future, it must also be applicable to CMOS circuit designs. The initial goal thus became to design a program which could accommodate both NMOS and CMOS technologies. As the design of SOLO progressed, it became clear that most of the methodologies and algorithms employed were indeed applicable to both NMOS and CMOS designs. The goal of a system applicable to both technologies was thus maintained.

3.1.1.3. Realization Constraints

The generality of a layout synthesis program also relates to the ability of that program to produce layouts which are compatible with different surrounding environments. Clearly a synthesis program which allows the user to specify constraints on the locations of external connections in a generated layout or to specify a desired aspect ratio will be of more use than one which creates layouts without any consideration as to the environments in which the generated cells will reside.

All schematic to layout systems in the literature allowed the positions of external connections to be specified. This seemed to be the minimal level of realization control needed to allow useful layouts to be generated. Discussions with designers also indicated that at least some control over the dimensions or aspect ratios of generated layouts was necessary in order to allow a user to guide the program towards a layout solution geometrically compatible with a surrounding environment [13].

SOLO was thus initially designed such that the positions of external connections could be specified along with the desired aspect ratio of the layout. As the development of the program continued, the set of possible realization constraints was expanded to provide users with additional control over the exact form of generated layouts. A description of the complete set of realization constraints is given in chapter 4.

3.1.2. Computational Efficiency

An automated layout program is only useful if it provides a significant design time advantage over human designers. One of the anticipated uses of SOLO was as a tool to allow users to quickly explore large design spaces including many different circuit topologies and transistor sizings. It was clear that the usefulness of the program in this role was directly related to its computational efficiency.

The computational goal therefore became to create a program that would run for no longer than a few CPU minutes for a circuit of reasonable size (roughly 20 transistors). In terms of the other schematic to layout programs presented in chapter 2, such a system would be more computationally efficient than DUMBO, Koss's program or TOPOLOGIZER. While it was not expected that the program would be as fast as Sc2 or the IBM program, it was hoped that it could be competitive with TALIB. This level of computational efficiency would allow the program to be applied to circuits including up to roughly 200 transistors while still producing layouts in a reasonable amount of time.

3.1.3. Layout Efficiency

A schematic to layout system can generate high performance layouts because it allows the circuit topologies and transistor sizings best suited to particular design situations to be employed. The performance advantage associated with being able to implement arbitrary circuit topologies is lost however if those topologies cannot be implemented efficiently. It was thus clear that in order for the system to be at all useful it would have to generate layouts that were superior in performance to those generated by conventional restricted topology layout systems. The hope was that the layouts would in fact be competitive with those created by human designers.

Although the layouts generated by TALIB were competitive with those of human designers (at least for some classes of circuits), TALIB was only applicable to NMOS circuits. On the other hand, the layouts generated by the programs applicable to CMOS circuits were necessarily of a lower quality due to the restrictive nature of the employed layout methodologies. The goal became to create a program which would generate layouts comparable in quality to those generated by TALIB but which was applicable to CMOS circuits.

3.2. System Strategy

In this section a summary of the overall strategy employed by SOLO is presented. This strategy was developed to provide a means by which the various system requirements described above could be satisfied.

3.2.1. Programming Methodology

The level of computational efficiency desired indicated that an algorithmic rather than a knowledge based programming methodology would be appropriate. Although TALIB managed to run quickly while employing a knowledge based approach, much of its computational efficiency could be attributed simply to well optimized code and to a fast implementation of the associated production language rather than to any fundamental advantage of the approach.

The goals of this research included the development of algorithmic methodologies for the automated layout of MOS circuits. While the development of an expert system for automated layout would provide insight into efficient layout methodologies for MOS circuits, such an approach would not provide similar insight into the development of algorithms appropriate for the implementation of such layout methodologies. An algorithmic programming methodology was thus selected.

3.2.2. Layout Methodology

The layout strategy that evolved from the set of system requirements includes two phases.

1. The circuit is first partitioned into sets of closely connected transistors which are mapped into layout. The partitioning and mapping into layout should allow local layout optimizations such as connection by abutment to be incorporated into the layout and should generate sublayouts that can effectively and easily be positioned by the subsequent general placement phase.
2. The generated sublayouts are then positioned and oriented using a general placement methodology.

The development of this two staged strategy was based on a number of considerations.

The circuit partitioning phase was incorporated to serve two purposes. First of all, the computational efficiency of SOLO could be improved because of the added level of hierarchy. Since the transistor placement portion of the program was now split into two phases, the program could run more rapidly when applied to large circuits. In addition, this partitioning could allow the layout efficiency of SOLO to be increased. Although local layout optimizations are difficult to incorporate into a general placement program, such optimizations could be incorporated during the more restricted mapping of the partitioned subcircuits into layout. These local layout optimizations would improve the quality of the generated layouts both in terms of area and electrical performance.

The placement methodology was selected because it seemed that a general layout methodology would be needed if high quality layouts were to be generated from arbitrary schematics. General methodologies were employed by both DUMBO and TALIB, the schematic to layout systems most successful in generating layouts competitive with those created by human designers. It was anticipated that such an approach might be difficult however since general layout methodologies had rarely been applied to the automated layout of CMOS circuits. The issues of well and substrate contact placement had always made more restrictive methodologies more appropriate for CMOS layout. This problem was addressed by incorporating into the placement phase of the program measures related to the suitability of a given transistor configuration to subsequent well and substrate contact additions. These measures allow a general placement methodology to successfully be applied to CMOS circuits.

3.2.3. Restructuring of the Hierarchy

One of the essential aspects of the layout synthesis approach presented in this thesis is a restructuring of the circuit description hierarchy into a new hierarchy more well suited to the physical domain. This restructuring is intended to help avoid the problems associated with inappropriate layout hierarchy typical of many automated layout systems.

While it was clear from the beginning of the design process that the overall strategy for layout would include such a restructuring of the hierarchy, it was not initially clear how this restructuring should be accomplished.

Since the new description hierarchy was to be appropriate for the layout domain, an effective method for restructuring the description hierarchy would have to be dependent at least to some extent on the target layout methodology. The selected strategy for restructuring thus included an initial phase during which the input circuit would be partitioned into subcircuits of closely connected transistors appropriate to be mapped into the leaf sublayouts of the target layout methodology.

Although the hierarchy corresponding to a circuit partitioned in this manner would be well suited to the physical domain, it was clear that this hierarchy could be improved by recognizing that some of the generated subcircuits were more closely related than others. Those subcircuits which compose a complex CMOS gate are closely related for example. A second restructuring phase during which closely related subcircuits would be clustered was therefore incorporated.

Finally, despite the fact that the hierarchy associated with the input circuit description would in many ways be inappropriate for the physical domain, it was clear that some aspects of that hierarchy could be incorporated into the new hierarchy in such a way as to improve the quality of the restructured hierarchy.

The complete strategy for restructuring thus included three stages and is summarized below.

1. The circuit is partitioned into subcircuits of closely connected transistors of similar size.
2. Closely related subcircuits are grouped.
3. The input hierarchy is preserved to the extent that it does not conflict with these restructurings.

3.2.4. Symbolic Layout as Output

Very early in the design process, the decision was made to generate symbolic layout as the output of the synthesis process. TALIB was the only know schematic to layout system which mapped circuits directly into mask level layouts. All of the other systems mapped schematics into symbolic layouts. The generation of symbolic layout as output had a number of anticipated advantages.

- Perhaps foremost, the use symbolic layout would provide a level of abstraction through which the SOLO could be isolated from the details of mask level layout. The ability to work with wires and transistors rather than with rectangles would reduce the complexity of the program substantially.
- Efficient compactors existed for symbolic layouts and thus SOLO could be somewhat more lax during the layout process than might otherwise be the case.
- The generation of symbolic layout as output would give the program the design rule independence that was required.
- Finally, the pitch matching capabilities of symbolic compactors provided a convenient means whereby desired alignments between external connections of generated cells could be maintained through the transformation to mask level layout.

3.3. Program Structure

The system strategy outlined in the previous section was developed into a program which performs an ordered sequence of operations leading to the generation of a final layout. This section presents a summary of the different phases the program.

1. **Specification Language**

   The input to SOLO is generated using an associated circuit specification language. The language allows the user to hierarchically describe the structural topology of the circuit as well as to specify to a limited extent how that circuit should be realized. These realization constraints allow the user to specify the exact sizes and types of transistors, the desired aspect ratio of the generated layout, and the relative positions of the external connections to the circuit.

2. **Restructuring of Hierarchy**

   The circuit description hierarchy is restructured into a new topologically based hierarchy.

3. **Mapping of Leaves into layout**

   The subcircuits which constitute the leaves of the new hierarchy are mapped into symbolic layout.

4. **Placement**

   The sublayouts are positioned and oriented using a general placement technique.

5. **Routing**

   The routing between the sublayouts is completed using conventional routing techniques. This routing is also represented as symbolic layout.

6. **Optimization**

   The layout is optimized by removing unconnected wires, removing unnecessary layer changes, and by maximizing the amount of metal interconnect.

7. **Compaction**

   Finally, the symbolic layout is compacted and a mask level layout description is generated.

Chapter 4
Circuit Specification Language

4.1. Introduction

SOLO includes a specification language which allows the user to hierarchically describe the structural topology of a circuit as well as to specify to a limited extent how that circuit should be realized. The schematic for a simple CMOS exclusive-or circuit is shown in figure 4-1. A corresponding specification language description of that circuit follows.

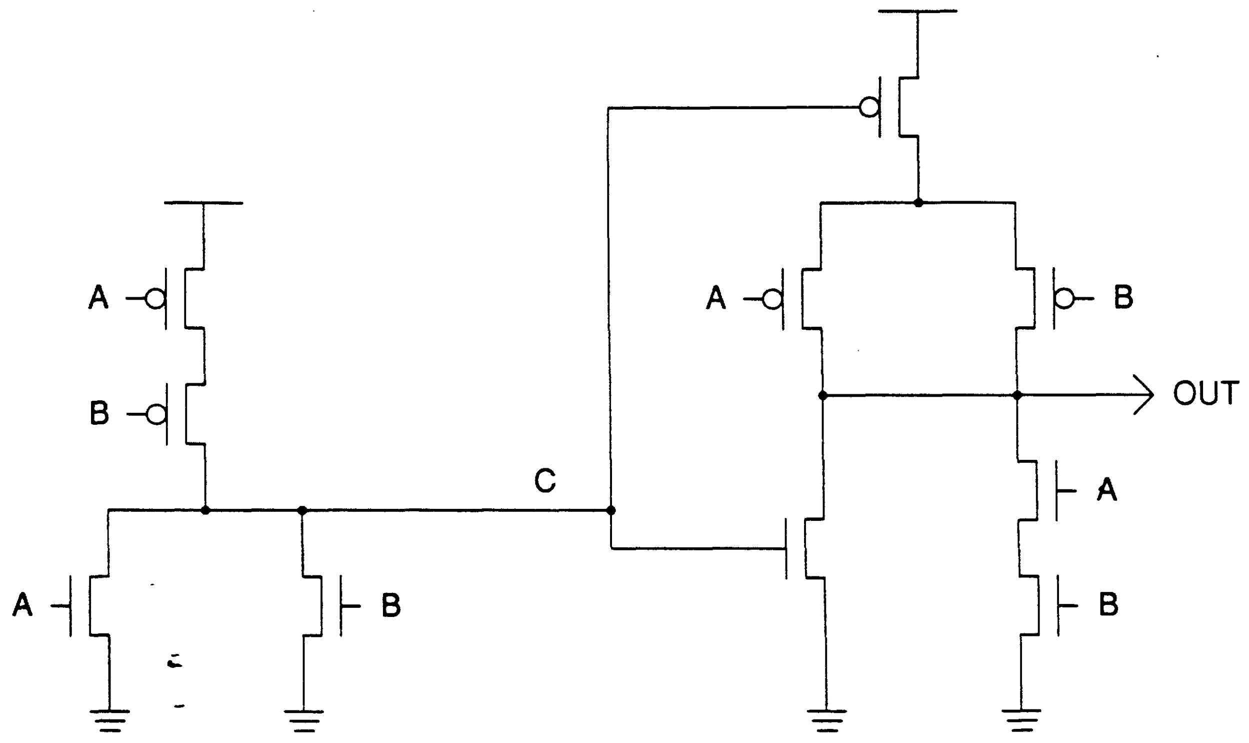

Figure 4-1: A Simple CMOS Circuit

```
(circuit-macro nor (a b out)
               (series vdd out (a b)
                       :type p :width 24 :length 3)
               (parallel out gnd (a b)
                         :type n :width 8 :length 3))


(defschematic xor (a b out)
  (local c)
  (set-technology cmos21)
  (left a b)
  (right out)
  (set-aspect-ratio 2 1)
  (nor a b c)
  (series vdd out
          ((transistor (c) :type p :width 24 :length 3)
           (parallel (a b) :type p :width 24 :length 3)))
  (series out gnd (a b) :type n :width 16 :length 3)
  (transistor out gnd (c) :type n :width 8 :length 3))
```

This example illustrates two different uses of the specification language.

Primarily, the language provides a means by which a desired circuit topology can be specified. Topological information can be represented through the use of several different constructs. Topology is conveyed through the use of circuit macros, through the specification of series and parallel connections of transistors, and through the specification of individual transistors. Different means for describing the same circuit are provided to simplify the task of the user in specifying a circuit and more importantly to allow some degree of flexibility to be introduced into the topological specification.

The specification language also provides a means by which constraints on how a desired circuit is to be realized can be introduced. In the example shown, constraints are given which specify the sizes and types of transistors, the sides on which external connections should be positioned, and a desired aspect ratio of the generated layout.

4.2. Language Capabilities

In this section the various capabilities of the specification language are outlined in more detail.

4.2.1. Specification of Circuit Topology

The primary purpose of the language is to allow the topological structure of a given circuit to be described. The topological specification of a circuit defines how the transistors in that circuit are interconnected as well as the exact types and sizes of those transistors.

The structure of a circuit can be described using a number of different constructs.

4.2.1.1. Transistors

The most straightforward means by which a given circuit can be described is to simply specify the connections of each transistor in the circuit individually. This is the technique used to describe circuit networks for the circuit simulation program SPICE [14]. While this method of circuit specification can become quite tedious for human designers to use when describing circuits of any size, it does have the advantage that such a definition can easily be derived from the network description of a circuit. This capability is quite important if SOLO is to be used as one stage of a larger silicon compiler system.

4.2.1.2. Series/Parallel Networks

This language also allows series and parallel connections of transistors (and of other series-parallel networks) to be specified. Since series-parallel subcircuit topologies occur often in digital circuits, describing a circuit in terms of series and parallel connections can often greatly simplify the task of a designer in specifying a given circuit topology.

This method of circuit description has the added advantage that it allows the user to specify a set of circuit topologies any one of which is acceptable rather than constraining the program to use one specific topology when such a restrictive specification is not necessary. The specification of a group of transistors in series whose exact ordering is not important is an example of how this construct may be used to specify a set of acceptable topologies. SOLO can often take advantage of this flexibility by delaying the binding of the exact topology of the circuit until the layout stage and in so doing generate more compact and efficient layouts.

4.2.1.3. Circuit Macros

Circuit macros are used to allow hierarchy to be introduced into the network description of a circuit. This hierarchy aids the designer in describing large circuits which include repeated subcircuits (such as logical gates). In addition, the structural hierarchy created using circuit macros serves as an aid in generating an appropriate physical hierarchy for use in describing a final layout.

4.2.2. Physical Realization Constraints

The specification language also allows some constraints on the exact form of the physical realization of a given circuit to be introduced.

4.2.2.1. Transistor Specification

The exact type and size of each transistor in the topology can be specified. This allows SOLO to be applied to optimized circuit schematics. The exact transistor sizings specified in an optimized schematic will be preserved in the generated layout.

4.2.2.2. Critical Node Information

The user can assign relative weightings to the nets of a circuit description. Nets along a critical delay path can be given higher weights than those whose capacitances are less critical. The weighting information is used by SOLO during the placement phase and has the effect of reducing the capacitances of the more highly weighted nets.

The weighting of nets can also be used simply to help guide SOLO towards some desired layout configuration.

4.2.2.3. Specification of Interface Constraints

The language also allows the user to constrain the locations of the external connections to the generated layout.

The placement of external connections can be constrained in a number of ways.

1. The side(s) on which a given external connection is to be located can be specified.
2. The exact ordering of the the connections on a given side of the layout can be defined.
3. Connections can be constrained to be positioned such that they will be

aligned (in one dimension) with appropriate connections on the opposite side of the cell. In this way cells can be generated which can be placed in an array and connected by abutment.

4.2.2.4. Aspect Ratio

The desired aspect ratio of the generated layout can be specified by the user. This specification is used to guide the placement phase of the program.

4.2.2.5. Feedthroughs

Finally, feedthrough nets can be specified. Such nets will have two external connections at the same position on opposite sides of the layout. These two connections are guaranteed to be connected within the cell in metal.[2] This construct thus provides a means by which clock lines and other nets which must be distributed to many cells may effectively be incorporated into generated layouts.

4.3. Language Syntax and Semantics

A detailed summary of the syntax and semantics of the specification language is given in appendix A.

[2]Since SOLO assumes that only a single layer of metal is available, feedthroughs are positioned such that they do not conflict with power and gnd lines.

Chapter 5
Layout Methodology

A system for mapping arbitrary MOS schematics into layout requires a layout methodology that can effectively be applied to many different classes of MOS circuits. A two staged strategy for layout was developed in an effort to provide such a generally applicable layout methodology. This layout strategy is summarized again below.

1. The circuit is first partitioned into sets of closely connected, similarly sized transistors which are mapped into layout. The partitioning and mapping into layout should allow local layout optimizations such as connection by abutment to be incorporated into the layout and should generate sublayouts that can effectively and easily be positioned by the subsequent general placement phase.
2. The generated sublayouts are then positioned and oriented using a general placement methodology.

In this chapter, the exact methodology used to partition circuits and to map the resulting subcircuits into layout is developed.

5.1. Gate Matrix Implementation of CMOS Circuits

Uehara and vanCleemput have presented a layout paradigm for efficiently mapping complex circuits into layout [15]. The methodology presented in their paper implements CMOS circuits on a linear array of transistors in a manner that maximizes abutment between transistors and in so doing minimizes the size of the array. The usefulness of the technique is somewhat limited however because it can only be used to implement CMOS circuits composed of series-parallel pull-up and pull-down trees. It is further required that the p-MOS and n-MOS halves of the circuits be duals.

The basic layout strategy involves the placement of transistors in an array configuration as is shown in figure 5-1. Each cell consists of a row of n-channel and a row of p-channel transistors. Since the pull-up and pull-down networks must be duals, there is a one to one correspondence between each n-channel transistor and an associated p-channel transistor sharing the same gate. By requiring that the n-channel and p-channel devices sharing a common gate be aligned vertically, the space required for making interconnections between the two rows can be minimized.

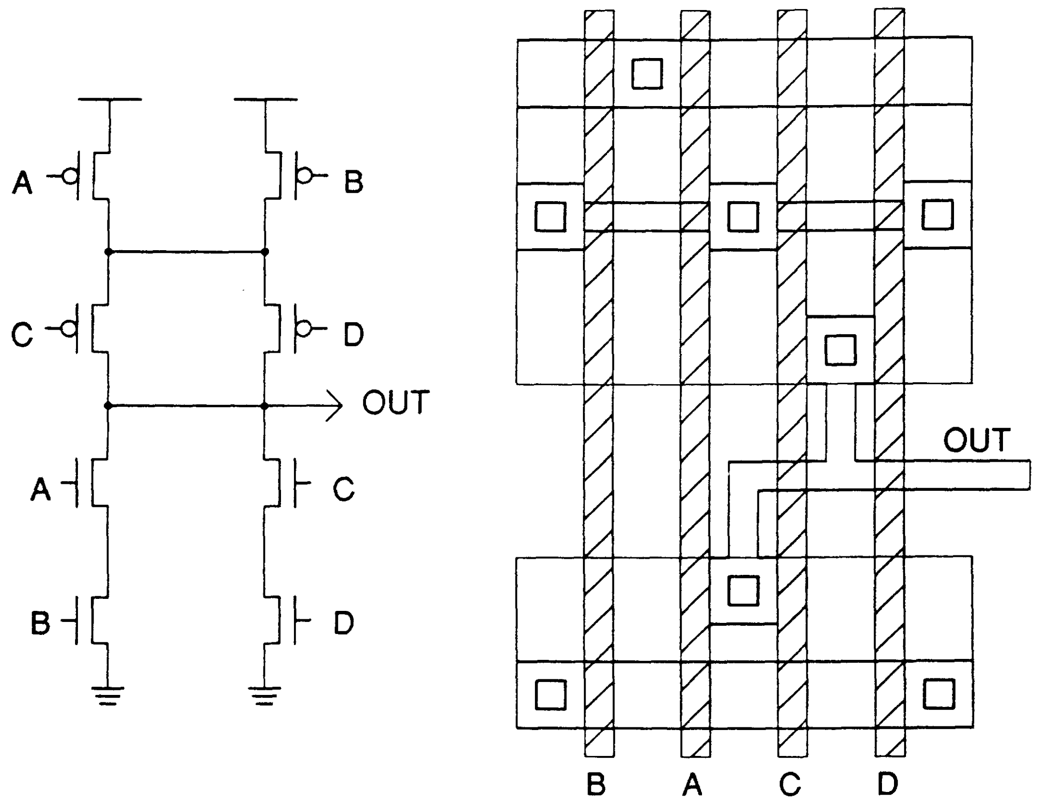

Figure 5-1: Gate Matrix Implementation of a Classical CMOS Gate

The area of each gate matrix module is proportional to the sum of the number of transistors in the module and the number of *diffusion breaks* required between chains of connected transistors. A break is required when there is no connection between adjacent transistors as is illustrated in figure 5-2. Thus, for a given number of transistors, the size of a module can be minimized by minimizing the number of required diffusion breaks.

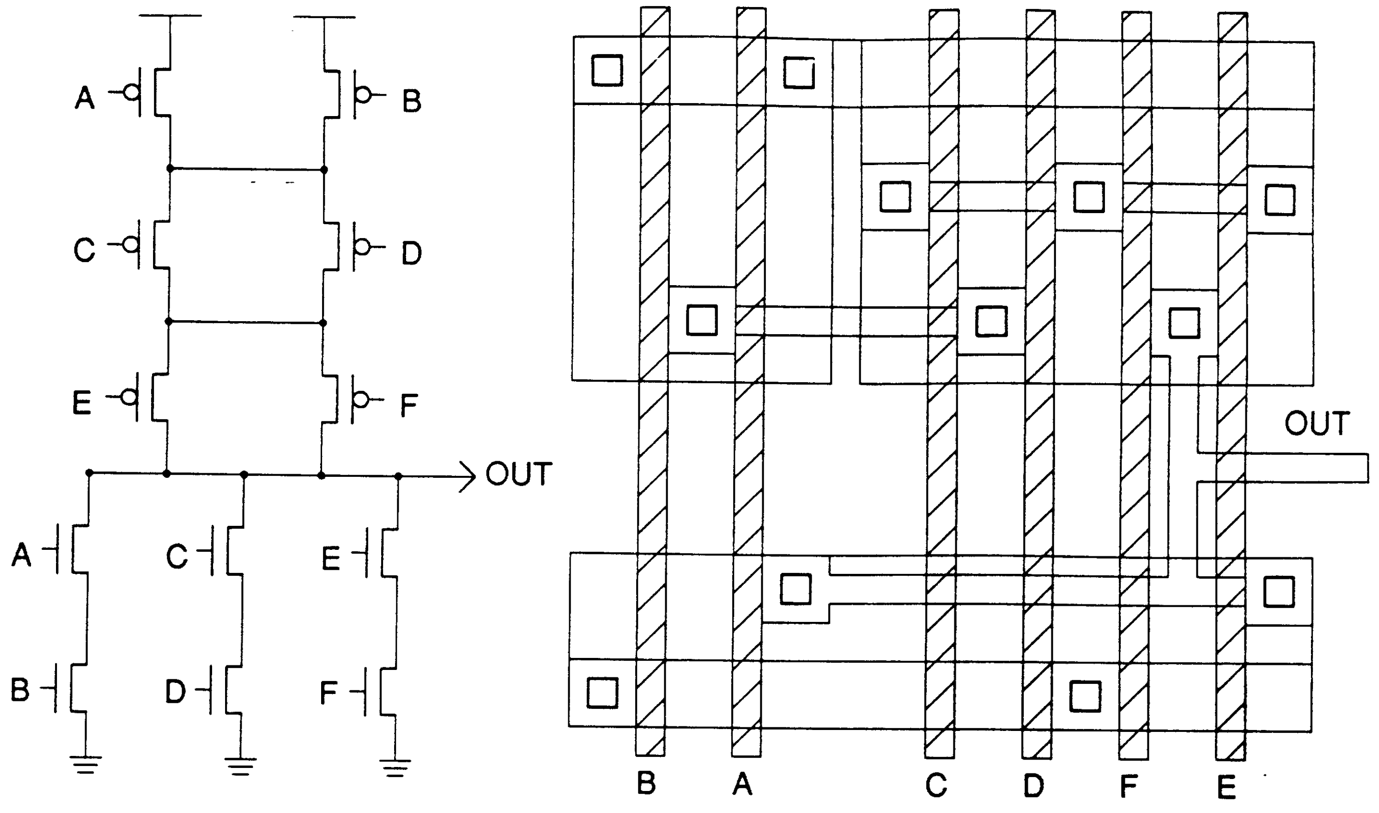

Figure 5-2: Gate Matrix Module with a Required Diffusion Break

5.1.0.1. A Circuit Graph Model

A circuit graph model is used by Uehara to determine a placement of transistors that will minimize the number of required diffusion breaks. One circuit graph is defined for the p-MOS half of the circuit while another is defined for the n-MOS half. Each graph is constructed as follows.

1. The drains and sources of transistors are represented by vertices in the graph.
2. Each transistor is represented by an edge between the vertices corresponding to the source and drain of that transistor.
3. Each edge in the graph is labeled with the net associated with the gate of the corresponding transistor.

A sample circuit and the corresponding circuit graphs are shown in figure 5-3.

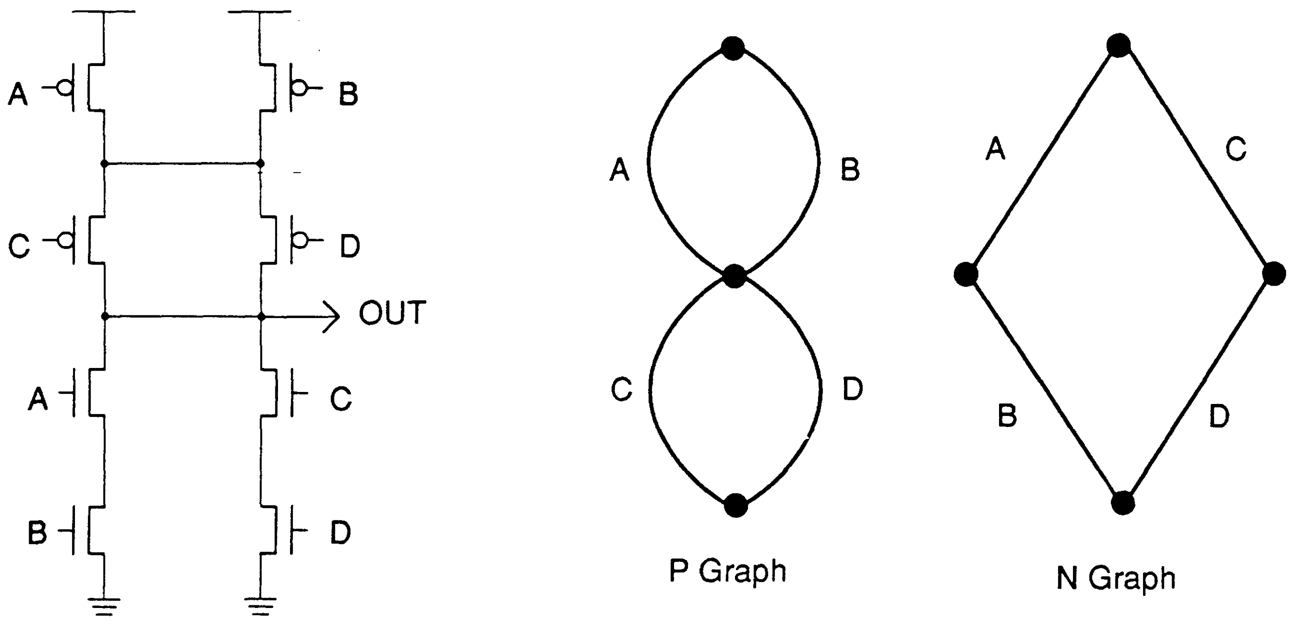

Figure 5-3: A Sample Circuit and Corresponding Circuit Graphs

Given this graph representation of a circuit, there is one property that is of particular interest in trying to minimize the number of required diffusion breaks.

If two edges share a vertex in the circuit graph, then their corresponding transistors may be connected by abutment.

Thus, in order to minimize the number of required diffusion breaks, it is necessary to find a set of paths[3] which is minimal in number and which covers all of the edges in the graph model. Furthermore, if the graph contains an Euler path [16] (i.e. a single path which covers all the edges of the graph), then the corresponding circuit can be laid out with no required diffusion breaks.

In order to allow the vertically aligned transistors in the gate matrix array to share a common gate, it is necessary to find covering paths on the two graph models with the same sequence of labels[4]. This will ensure that n-MOS and p-MOS transistors sharing a common input will have the same horizontal position in the array.

[3]A path is a sequence of edges where the final vertex of one edge is the initial vertex of the next.

[4]In other words, the sequence of labels associated with the sequence of edges in each covering path should be identical.

The optimal layout algorithm can thus be summarized as [15]:

1. Find all Euler paths in each graph.
2. Find p-graph and n-graph Euler paths with identical labelings.
3. If 2 is not found, then decompose the graphs into a minimal number of subgraphs such that 2 can be achieved by separate Euler paths.

5.1.1. Limitations of the Gate Matrix Technique

Although the gate matrix layout methodology can be used effectively to realize the specific types of circuits described above, the technique has several limitations when it is applied to more general circuit methodologies.

5.1.1.1. Sized Transistors

The gate matrix technique is intended for use with identically sized transistors. When different size transistors are used, the height of the transistor array is dictated by the widths of the widest n-MOS and p-MOS transistors. If a circuit includes transistors that are significantly varied in size, a gate matrix realization can be inefficient in the way of layout area. This is illustrated in figure 5-4.

5.1.1.2. Non-Standard Circuit Methodologies

The layout technique presented in [15] is only applicable to a very limited subset of MOS circuits. More specifically, the technique requires that n-MOS and p-MOS halves of the associated circuit graph be duals and that the circuit be composed entirely of series-parallel combinations of transistors.

While techniques for realizing more general MOS circuits using a gate matrix methodology do exist [10] [9], the usefulness of the gate matrix technique is inherently limited to circuit methodologies which include equal numbers of n-MOS and p-MOS transistors. The use of the gate matrix technique to realize circuit methodologies which include unequal numbers of n-MOS and p-MOS transistors (such as domino CMOS) results in much wasted area due to the unused transistor "slots" which necessarily occur. A gate matrix realization of a domino CMOS circuit illustrating this inefficiency is shown in figure 5-5.

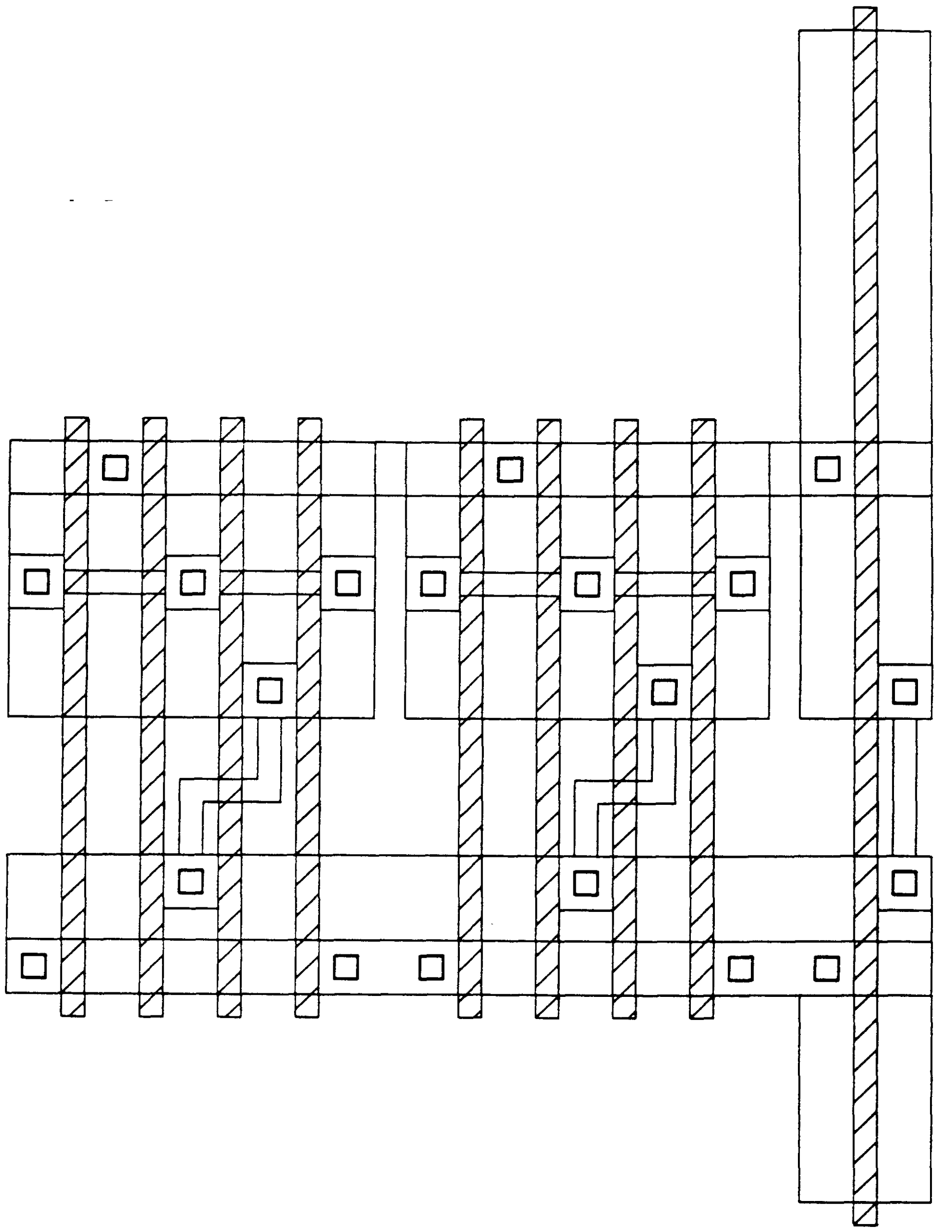

Figure 5-4: Gate Matrix Realization Including Sized Transistors

5.1.2. Transistor Splitting

One means by which the usefulness of the gate matrix technique can be extended to more classes of MOS circuits is through the use of transistor splitting. This technique allows sized transistor circuits to be efficiently realized using a gate matrix methodology. In this technique, very wide transistors are split into several smaller transistors connected in parallel. By properly selecting which transistors to split as well as the sizes of the new transistors, the total area of the transistor array can be minimized. This

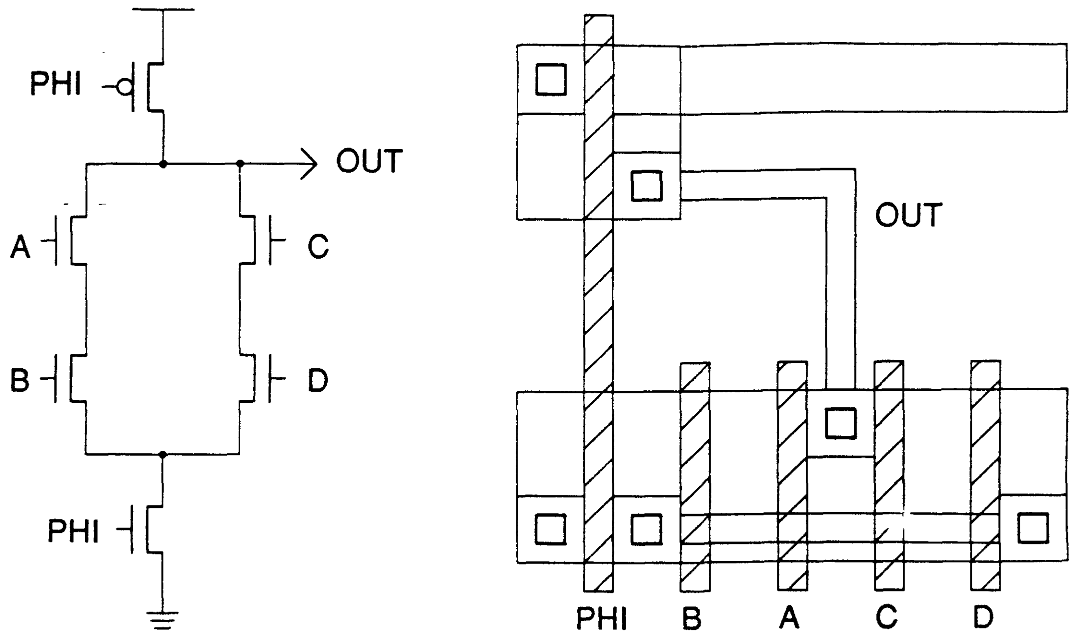

Figure 5-5: Gate Matrix Realization of a Domino CMOS Circuit

technique is used by the layout synthesis program Sc2 [9] and allows that program to effectively be applied to sized transistor circuits.

An example of the use of the transistor splitting technique is shown in figure 5-6. This figure illustrates a more efficient realization of the same layout as was shown in figure 5-4.

Transistor splitting still does not allow the gate matrix layout methodology to be efficiently applied to all MOS circuit methodologies however. Although sized transistor circuits can be realized in a more efficient manner, this technique does nothing to help in the realization of non-standard circuit methodologies. Circuits must still include equal numbers of n-MOS and p-MOS transistors if the gate matrix layout technique is to be effective.

5.2. General Placement of Gate Matrix Modules

In this section a general layout methodology based on a variation of the gate matrix technique is presented.

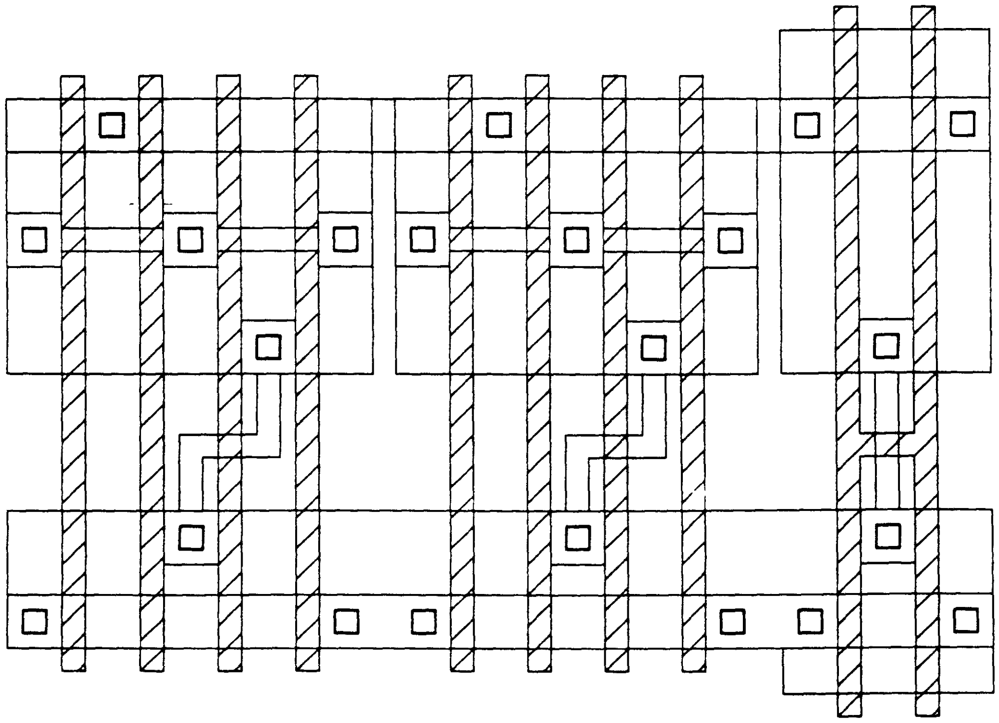

Figure 5-6: Gate Matrix Realization with Transistor Splitting

5.2.1. Gate Matrix Mapping of Subcircuits into Layout

It is clear that even with the use of transistor splitting the gate matrix technique cannot be applied *effectively* to MOS circuits in general. The technique does provide a means by which partitioned subcircuits can efficiently be mapped into layout however. If subcircuits are constrained to only include one type of transistor, then they can be realized on a single row using the gate matrix technique. This is in contrast to the gate matrix examples that have been shown in which two rows (one for n-MOS transistors and one for p-MOS transistors) have been used. By limiting the use of the gate matrix technique to subcircuits which can be realized on a single row, constraints on the number of p-MOS and n-MOS transistors can be eliminated and non-standard circuit methodologies can be implemented efficiently.

The gate matrix technique has a number of other desirable attributes that also make it well suited to the mapping of subcircuits into layout. It allows connections by abutment to be incorporated into the layout thereby improving layout efficiency. Furthermore, given that the included transistors are similarly sized the generated sublayouts tend to be rectangular and thus can effectively be positioned by a subsequent general placement program.

5.2.2. Partitioning Strategy

Given a means for mapping appropriate subcircuits into layout, a methodology must be developed for dividing a circuit into such subcircuits. This partitioning strategy must incorporate a number of basic goals.

1. The grouping of transistors should in some sense relate to their *connectedness*. If this can be done in an appropriate manner, then the number of connections between the different groups will be minimized and a *modular* description hierarchy will result.
2. The partitioning methodology should generate groups that include enough transistors that the use of the selected groups of transistors as leaf modules rather than individual transistors will significantly reduce the number of objects to be placed and interconnected.
3. The partitioned groups of transistors should constitute subcircuits which can easily and efficiently be mapped into the physical domain using the target single-row gate matrix methodology.

Before any circuit partitioning can be accomplished, it is necessary to create an abstract representation of the circuit topology which captures all the information relevant to the partitioning process. A circuit graph model such as that defined by Uehara [15] is appropriate for this purpose.

Given a circuit graph description of a circuit, the problem of partitioning a circuit into appropriate groups of transistors becomes the problem of partitioning the circuit graph into a set of appropriate subgraphs. One must now consider how the stated goals of the partitioning strategy can be incorporated into an algorithmic methodology for partitioning a graph.

The target single-row gate matrix layout methodology implements series chains of transistors in an optimal manner. A partitioning method aimed at dividing the circuit graph into series combinations of transistors might thus seem a reasonable strategy. This proves not to be the case however. While series subcircuits can efficiently be mapped into a gate matrix form, the series subcircuits extracted from typical MOS circuits tend to contain relatively few transistors. Long series chains of transistors are typically associated with poor circuit performance and thus such circuit constructs tend to be avoided. Partitioning the exclusive-or circuit of figure 4-1 into series combinations of transistors results in 8 series subcircuits. This can hardly be considered an effective partitioning of a circuit which contains only 10 transistors.

An effective partitioning strategy must be based on a more general circuit characteristic which yields larger subcircuits while at the same time continuing to ensure that the subcircuits can be efficiently realized using a gate matrix methodology.

The gate matrix methodology also realizes series-parallel combinations of transistors in an efficient manner and in fact the technique presented by Uehara is designed to implement just such circuits. A strategy based on the partitioning of a circuit into subcircuits composed of series-parallel combinations of transistors has several desirable properties. The subcircuits generated in this way tend to be larger than those generated using a simple series subcircuit partitioning scheme. Unlike long series chains of transistors, large series-parallel combinations of transistors are abundant in digital MOS circuits. Partitioning the same exclusive-or circuit of figure 4-1 using a series-parallel partitioning scheme results in only 4 subcircuits rather than the previous number of 8. In addition, the number of connections between series-parallel subcircuits tends to be quite low. Each series-parallel subcircuit can have at most one connection for each transistor gate along with two additional connections corresponding to the two distinguished vertices of the associated series-parallel subgraph. A series-parallel circuit partitioning will thus tend to be modular.

Although a strategy based on the partitioning of the circuit graph into series-parallel subgraphs appears promising, there are a number of associated problems that must be considered. First of all, while the mapping of series-parallel circuits into layout can only be efficient if the included transistors are of similar size, there is presently no guarantee that the transistors included in each subcircuit will indeed be of similar size. This problem must be addressed. In addition, an efficient method must be developed for partitioning a graph into a minimal number of series-parallel subgraphs.

The first problem can be addressed simply by modifying the partitioning strategy slightly so that a circuit is divided into series-parallel combinations of similarly sized transistors rather than simply into series-parallel combinations of transistors. The standard deviation of the widths of a set of transistors provides a reasonable metric as to how *similar* in size those transistors are. Although this added constraint occasionally results in the splitting of a series-parallel subcircuit into two or more smaller subcircuits, the sizes of generated subcircuits are not on the average decreased significantly.

In addressing the second problem, it is first necessary to consider the nature of series-parallel graphs.

A series-parallel graph consists of the series or parallel combination of other series-parallel graphs. In the degenerate case, a single edge joining two vertices forms a series-parallel graph. Associated with a series-parallel graph are two *distinguished* vertices to which other series-parallel graphs may be connected [17].

This recursive definition of series-parallel graphs can be translated into an efficient algorithm for partitioning a circuit into a minimal number of series-parallel subcircuits. The algorithm also ensures that the generated subcircuits each include similarly sized transistors.

The complete circuit partitioning algorithm is summarized below. This algorithm takes a circuit graph as input and returns a set of subgraphs each corresponding to an appropriately partitioned subcircuit.

1. Identify those vertices which correspond to Vdd, Gnd, transistor gates, and external connections to the circuit. Mark them as *special* vertices.
2. For each edge in the graph, create a subgraph consisting of that edge and the two connected vertices. Mark the two vertices as *distinguished* vertices.
3. For each subgraph:
   a. Find any other subgraph which shares the same two distinguished vertices.
   b. **if** the two subgraphs have corresponding transistors that are similarly sized[5]

      then Merge the two subgraphs into a single subgraph with the same two distinguished vertices.
4. For each vertex that is not special:
   a. Find all subgraphs for which this vertex is a distinguished vertex.
   b. **if** there are exactly two such subgraphs **and** the two subgraphs have corresponding transistors that are similarly sized

      then Merge the two subgraphs into a single subgraph. The distinguished vertices of the new subgraph will be the two distinguished vertices of the previous subgraphs that were not shared between those subgraphs.
5. Repeat steps 3 and 4 until no subgraph consolidations occur.
6. Return the set of subgraphs.

[5]A set of transistors is considered to be similarly sized if the standard deviation of the widths of those transistors is below some threshold level.

This algorithm provides an efficient means for partitioning a circuit into a minimum number of similarly sized series-parallel subcircuits. Since both the number of subgraphs and the number of vertices will be of the order of the number of transistors in the circuit, $O(T)$, this algorithm will run in $O(T^3)$ time.

A partitioning algorithm based on the division of circuit into series-parallel combinations of similarly sized transistors was thus selected. This methodology generates subcircuits which are modular and which can be efficiently mapped into gate matrix layout using straightforward techniques such as the one described in this chapter. Furthermore, since this method ensures that the transistors included in each subcircuit are similarly sized, the corresponding sublayouts tend to be rectangular with little wasted layout area.

5.2.3. The Layout Process

The overall layout process can be summarized as follows:

1. The circuit is partitioned into a minimal number of subcircuits each of which is a series-parallel combination of similarly sized, identically typed transistors. It is clear that such a partitioning is always possible since in the degenerate case each individual transistor will constitute such a subcircuit.
2. Subcircuits which are each other's duals are paired.
3. A technique similar to that presented by Uehara [15] is used to map each of these subcircuits into a single-row gate matrix module (which includes any required internal connections).

   In the case of subcircuits which are duals, it is ensured that both subcircuits are mapped into gate matrix layout in such a way that the gates associated with their transistors are identically ordered.
4. A general placement technique is used to establish the final positions and orientations of the generated sublayouts.

The mapping of a series-parallel subcircuit into a gate matrix module is illustrated in figure 5-7. This figure illustrates many of the advantages of this aspect of this layout technique.

Because the transistors included in the sample subcircuit are all of similar size, the generated layout is indeed rectangular with little wasted layout space. In addition, this sublayout is very wirable. Each external connection is accessible on at least two sides of the sublayout and some are accessible on more. In the example shown, the *OUT* node is accessible on all four sides of the generated module. Finally, since the gate matrix layout style tends to minimize parasitic capacitances, it is clear that this module will also be electrically efficient.

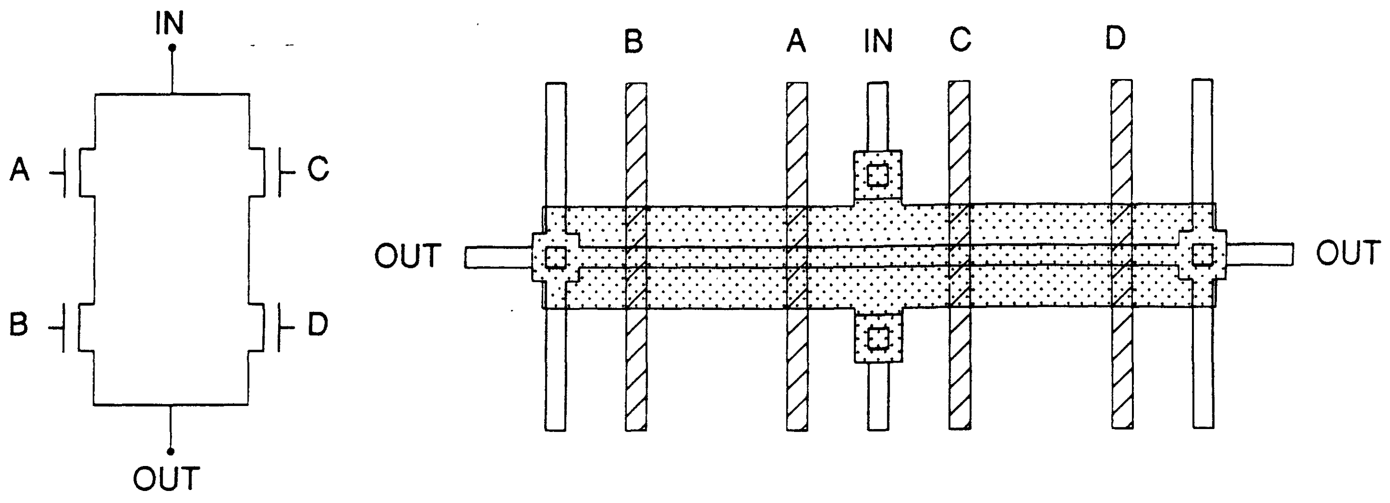

Figure 5-7: Mapping a Series-Parallel Subcircuit into a Gate Matrix Module

Since the methodology used for placing these generated gate matrix modules is general in nature, it allows even sublayouts which vary greatly in size to be effectively placed. Circuits including sized transistors can thus be efficiently realized using this layout technique. Figure 5-8 illustrates the application of this approach to a circuit including sized transistors.

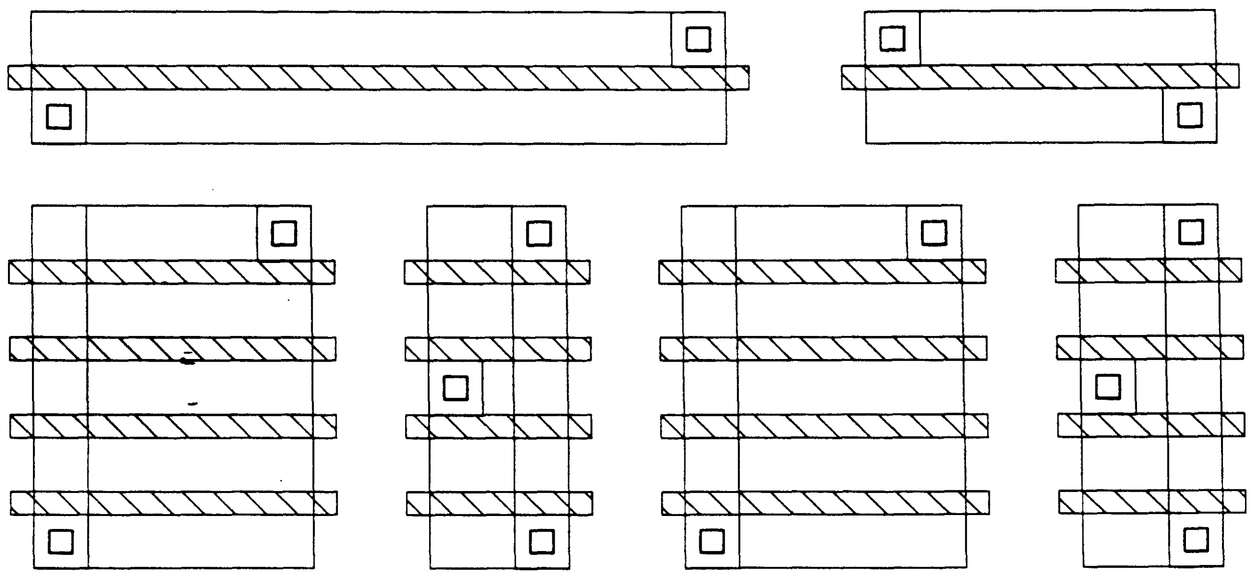

Figure 5-8: General Placement of Gate Matrix Modules

This technique also provides an efficient means for realizing non-standard circuit methodologies. An example in which this technique is used to realize the domino CMOS circuit presented earlier is shown in figure 5-9. The general placement methodology allows the p-MOS pull-up to be positioned next to the n-MOS pull-down tree rather than above it. If one considers a circuit similar to the one shown here but in which the clocked pull-up is replaced with a depletion load device, it becomes clear that this layout technique should also be effective in realizing NMOS circuits.

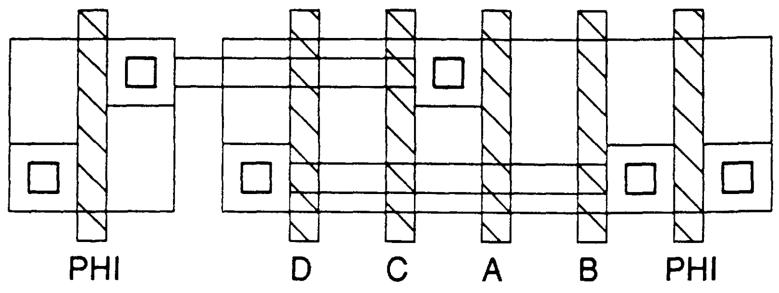

Figure 5-9: Efficient Realization of a Domino CMOS Circuit

The layout methodology presented here provides an effective means for mapping many different classes of MOS circuits into layout. The methodology allows sized transistor circuits as well as non-standard circuit methodologies to be efficiently realized. This layout methodology, based on the general placement of gate matrix modules, was thus incorporated into SOLO.

Chapter 6
Restructuring the Hierarchy

An essential aspect of the layout synthesis approach presented in this thesis involves a restructuring of the hierarchy used in describing a circuit into a new hierarchy more well suited to the physical domain.

The strategy which was developed to implement this restructuring involves three phases. During the first phase, the flattened circuit description is partitioned into subcircuits appropriate to be mapped into layout using the target single-row gate matrix layout methodology. This phase was included specifically to accommodate the selected layout methodology. In the second phase, closely related subcircuits are identified and grouped. This adds another level of hierarchy to the circuit description. The physical hierarchy is finalized by preserving those parts of the hierarchical structure of the original circuit description which do not conflict with the restructurings described above.

In this chapter the exact methodology by which these restructuring phases are accomplished is presented and a number of restructuring examples are shown.

6.1. Partitioning

The first stage of the restructuring involves a partitioning of the circuit into a number of subcircuits each of which will correspond to one leaf module in the generated layout hierarchy. The method by which this partitioning is implemented is clearly dependent on the target layout methodology. The target layout methodology involves the general placement of gate matrix modules corresponding to series-parallel combinations of similarly sized transistors. The restructuring that is accomplished during this phase is thus specifically the series-parallel partitioning that was developed in chapter 5.

6.2. Grouping of Closely Related Subcircuits

During the second stage of the restructuring closely related subcircuits are identified and grouped. There are two very specific types of groupings that are included in this phase of the restructuring.

1. Subcircuits which constitute the pull-up and pull-down trees of a common output node are paired.
2. Subcircuits which are a single n-MOS and p-MOS transistor respectively and which together form a CMOS pass gate are paired.

6.2.1. Grouping Pull-Up and Pull-Down Structures

In order to ensure that all possible pairings of pull-up/pull-down structures are accomplished, each subcircuit is compared with all other subcircuits to determine if the two form a pull-up/pull-down pair. During these comparisons the following rules are used to check if two subcircuits do indeed form a pull-up/pull-down pair and thus should be grouped.

1. The two subcircuits must share a common distinguished vertex.
2. Among the two remaining distinguished vertices, one must be connected to Gnd and the other must be connected to Vdd.
3. In the case where it is possible to pair a single subcircuit with more than one other subcircuit, the following additional criteria are used (in this order) to determine which subcircuits are paired. If more than one subcircuit satisfies the relevant criterion, then an arbitrary choice is made.

    a. Subcircuits which are duals are paired.
    b. If the contention is between a number of subcircuits one of which is a depletion-mode pull-up, then that subcircuit is selected.
    c. If the contention is between a number of subcircuits one of which appears to be a clocked pull-up[6], then that subcircuit is selected.

[6]A subcircuit *appears* to be a clocked pull-up if it consists of a single transistor and the net associated with the gate of that transistor has a name including any one of the strings "phi", "clk', or "clock"

6.2.2. Grouping Pass Transistor Structures

After pull-up/pull-down structures have been paired, a similar procedure is used to pair pass transistor structures. In this case however, the following pairing rules are used.

1. One subcircuit must be a single n-MOS transistor.
2. The other subcircuit must be a single p-MOS transistor.
3. The two transistors must be connected at both the drain and the source.

6.3. Preservation of Existing Hierarchy

The discussion thus far has centered only on the generation of new hierarchy. No mention has been made of how this new hierarchy is incorporated into the description hierarchy that is already present.

Each of the restructurings described above involves the introduction of another level of hierarchy through the grouping of some number of objects in the existing hierarchy. These restructurings can easily be incorporated into the existing hierarchy only if all of the grouped objects share a common parent. Therefore, in order to allow such restructurings to be accomplished, one additional type of restructuring is included. This restructuring effects changes in the hierarchy so that the objects to be grouped will share a common parent. More specifically, if a number of objects in the hierarchy are to be grouped and those objects do not share a common parent, then the hierarchy is altered so that each of those objects becomes a child of their closest common ancestor.

The third phase of the restructuring is thus not implemented as a single phase but is instead incorporated into each of the other restructuring phases. This aspect of the restructuring has the effect of limiting changes in the original hierarchy to only those necessary to allow the other restructurings to be accomplished. The original hierarchy is preserved to the extent that it does not conflict these other restructurings.

6.4. Restructuring Examples

In this section two restructuring examples are presented. In one example the original hierarchy is appropriate for the physical domain and thus the restructuring simply results in the addition of new hierarchy to the leaves of the original hierarchy. In the other example the original hierarchy is not appropriate for the physicai domain. The subsequent restructuring therefore effects real changes in the description hierarchy.

The same exclusive-or circuit as was used in chapter 4 will be used in these examples and is duplicated in figure 6-1. The circuit graph corresponding to that circuit is given in figure 6-2.

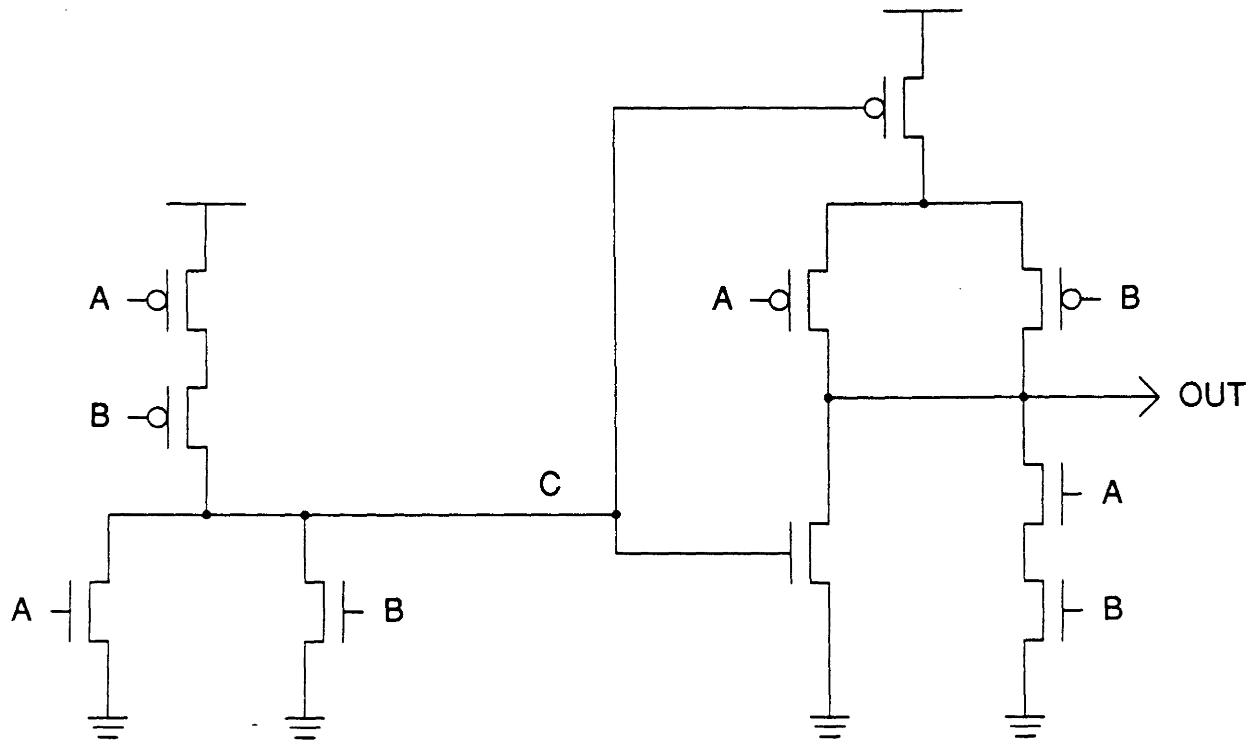

Figure 6-1: A Sample Circuit

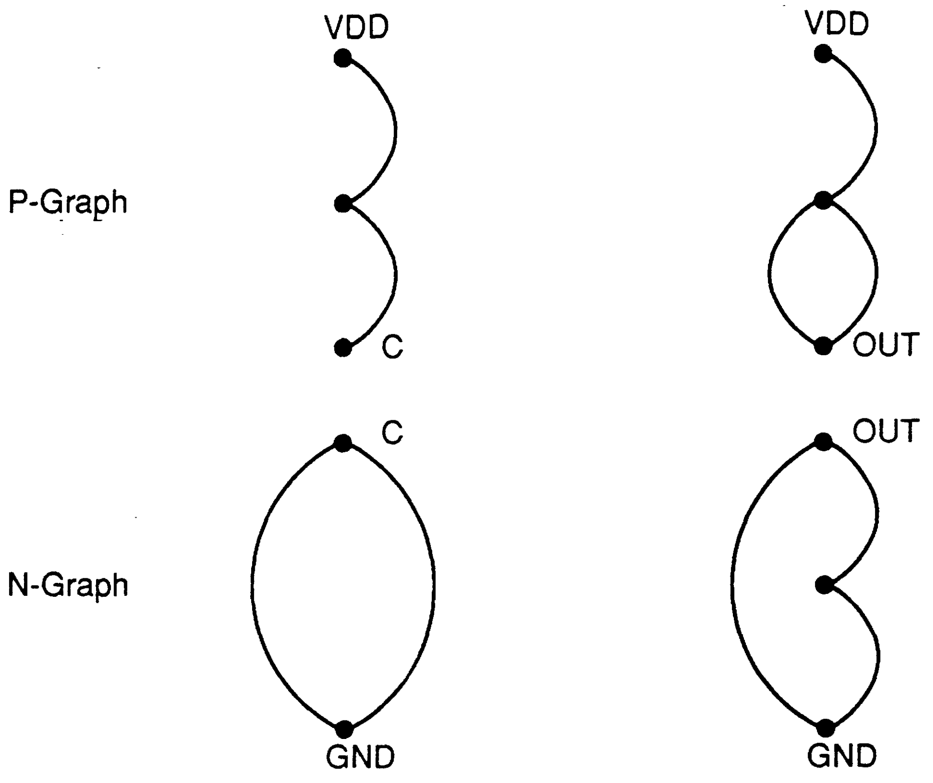

Figure 6-2: The Corresponding Circuit Graph

6.4.1. An Example with Appropriate Input Hierarchy

For this example it will be assumed that the circuit of figure 6-1 is defined as follows:

```
(circuit-macro nor (a b out)
               (series vdd out (a b)
                       :type p :width 24 :length 3)
               (parallel out gnd (a b)
                         :type n :width 8 :length 3))


(defschematic xor (a b out)
  (local c)
  (set-technology cmos21)
  (left a b)
  (right out)
  (nor a b c)
  (series vdd out
          ((transistor (c) :type p :width 24 :length 3)
           (parallel (a b) :type p :width 24 :length 3)))
  (series out gnd (a b) :type n :width 16 :length 3)
  (transistor out gnd (c) :type n :width 8 :length 3))
```

The hierarchical structure associate with this input description is shown in figure 6-3.

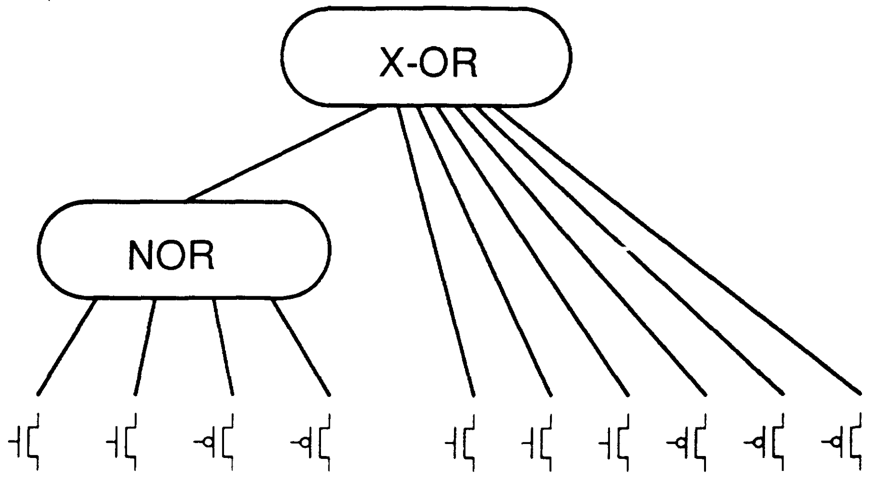

Figure 6-3: Original Circuit Description Hierarchy

Partitioning the corresponding circuit graph into a minimal number of series-parallel subgraphs yields the set of graphs illustrated in figure 6-4.

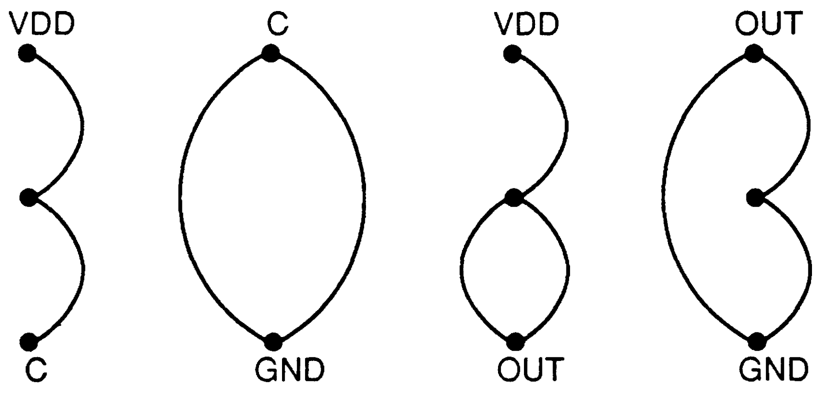

Figure 6-4: The Circuit Graph Partitioned into Series-Parallel Subgraphs

Incorporating the associate transistor groupings into the description hierarchy yields the structure shown in figure 6-5.

The next step in the restructuring process is to group associated pull-up and pull-down structures. Two pull-up/pull-down pairs are present in this circuit. Adding these

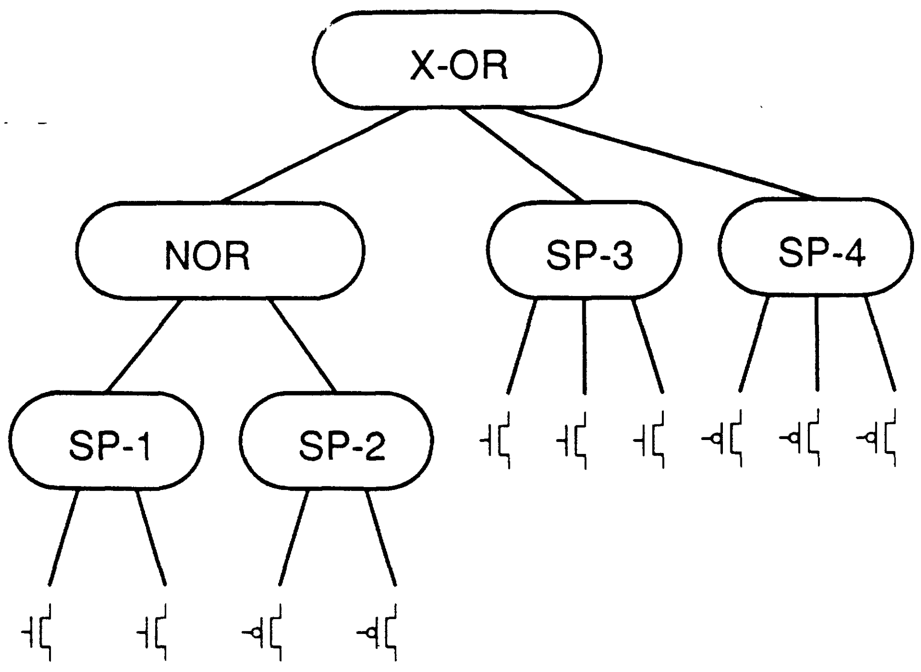

Figure 6-5: Intermediate Circuit Description Hierarchy

pairings to the circuit description hierarchy results in the final description hierarchy illustrated in figure 6-6.

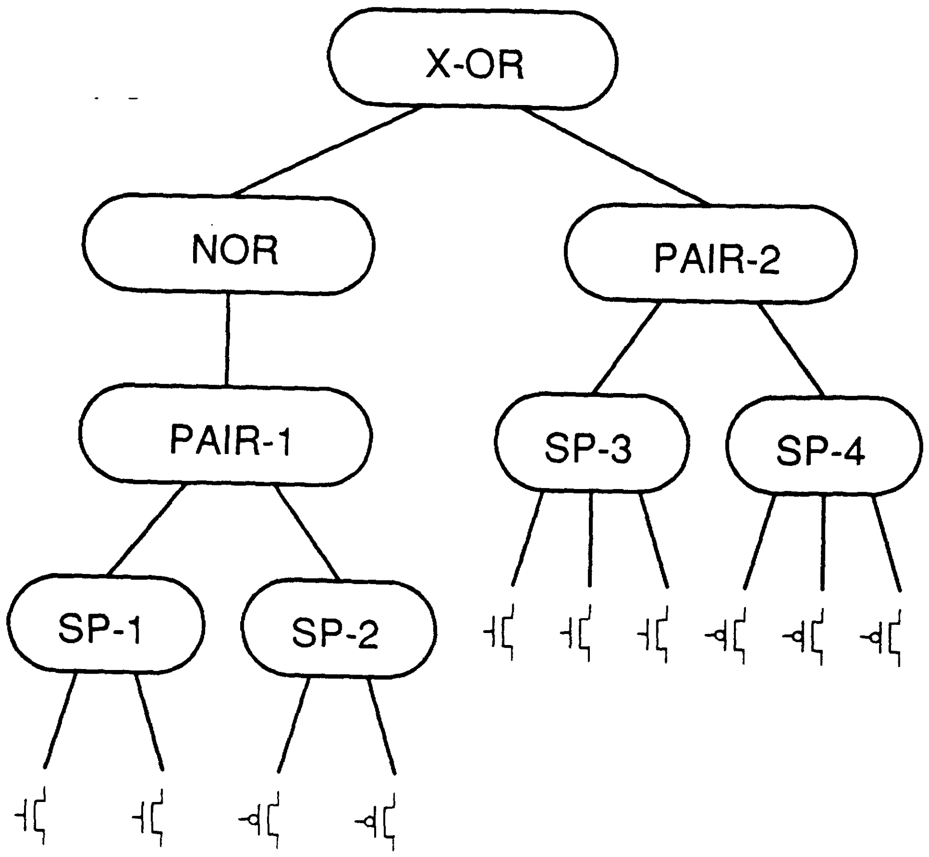

Figure 6-6: Final Circuit Description Hierarchy

6.4.2. An Example with Inappropriate Input Hierarchy

Assume for this example that the circuit of figure 6-1 had instead been defined as follows:

```
(circuit-macro p-trans (a b c out)
               (series vdd c (a b)
                       :type p :width 24 :length 3)
               (series vdd out
                       ((transistor (c)
                                    :type p :width 24 :length 3)
                        (parallel (a b)
                                  :type p :width 24 :length 3))))

(circuit-macro n-trans (a b c out)
               (parallel c gnd (a b)
                         :type n :width 8 :length 3)
               (parallel out gnd
                         ((series (a b)
                                  :type n :width 16 :length 3)
                          (transistor
                            (c)
                            :type n :width 8 :length 3))))

(defschematic xor (a b out)
  (local c)
  (set-technology cmos21)
  (left a b)
  (right out)
  (p-trans a b c out)
  (n-trans a b c out))
```

The hierarchical structure associated with this description is illustrated in figure 6-7.

Partitioning the circuit graph into series-parallel subgraphs results in the same groupings of transistors as in the previous example. Incorporating these groupings into this description hierarchy results in the structure shown in figure 6-8.

Once again, the next step is to group associated pull-up and pull-down structures. In this case however, the pull-up and pull-down structures to be paired do not share a common parent. The hierarchy is therefore altered by making each of the subcircuits to be paired a child of their closest common ancestor (in this case *XOR*). This gives the two subcircuits to be paired a common parent. After this hierarchy change and the incorporation of the appropriate pairings into the circuit description hierarchy, the final description hierarchy shown in figure 6-9 results.

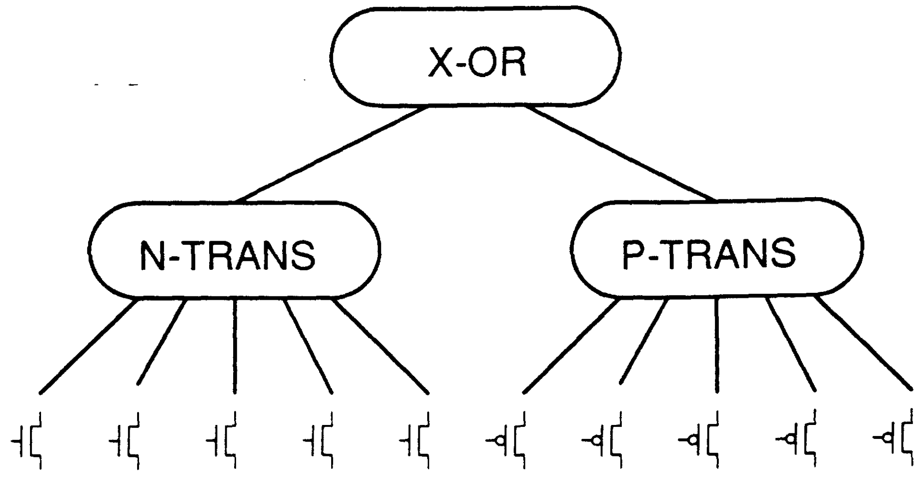

Figure 6-7: Original Circuit Description Hierarchy

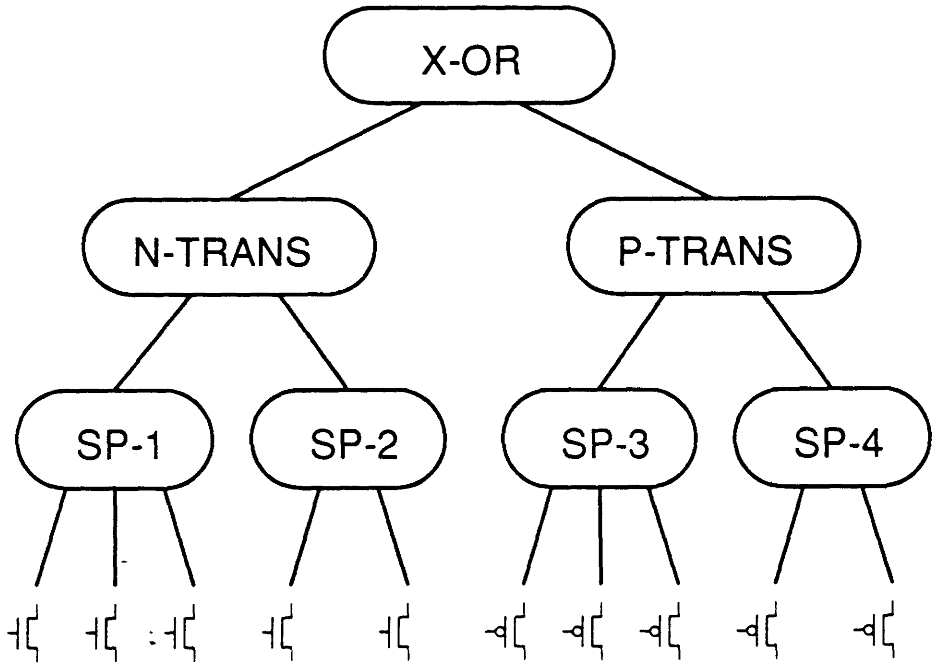

Figure 6-8: Intermediate Circuit Description Hierarchy

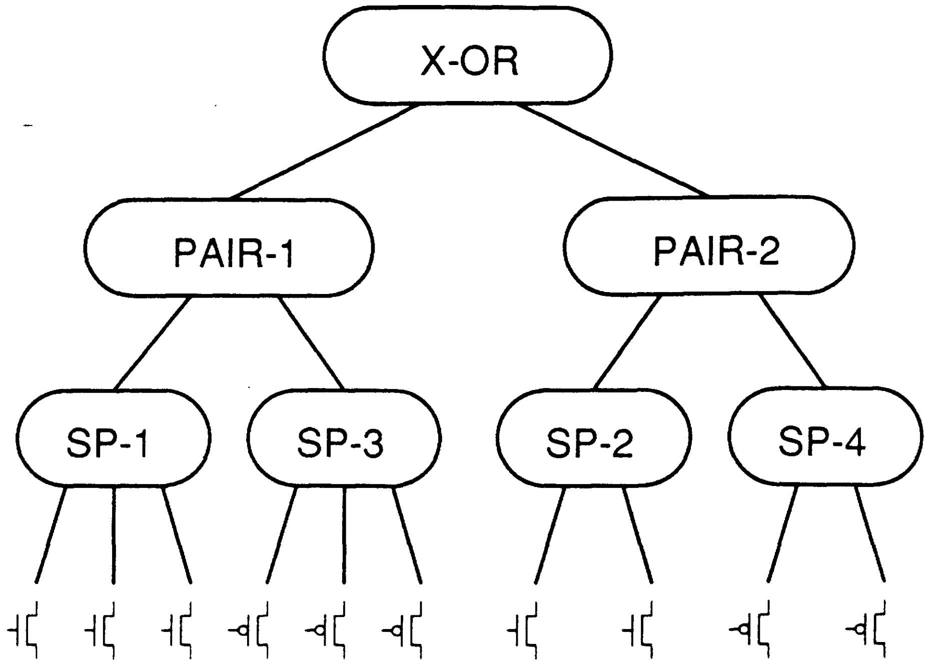

Figure 6-9: Final Circuit Description Hierarchy

Chapter 7
Placement

One of the primary goals of this research was to demonstrate that if the hierarchy of the physical description of a circuit design is generated in a proper manner then the subsequent placement and interconnection of the leaf sublayouts of that hierarchy can be accomplished using conventional placement and routing techniques without a substantial area or performance penalty over a true full custom design. If this was so, a system for generating efficient mask level layouts from circuit schematics could be implemented without the need to develop any new or novel placement and routing techniques.

The placement program for positioning the generated leaf sublayouts thus employs conventional placement techniques similar to those employed by other placement systems. This chapter outlines the program and describes the methodologies used in its implementation.

7.1. Overview

Although the program presented here is based in large part on techniques that have been used before, it is not a direct reimplementation of any one existing program. Its development was guided by the specific nature of this placement problem. More specifically, the program was designed to satisfy the following set of requirements.

- The placement program must be able to position arbitrarily sized rectangular modules in a general manner.
- The program must be hierarchical in nature. This allows it to make use of the circuit description hierarchy generated during the restructuring phase of SOLO.
- In order to ensure the quality of the generated placements, the program must base its placement decisions on the specific locations of terminals within a module as well as on other application specific cost functions.

The placement algorithms employed by the PI System [18] [19] [7] provided a framework by which a placement program meeting these requirements could be developed.

Consistent with the philosophy of that system, a placement program based on an initial rough placement phase and a subsequent *refinement* phase was designed. A more detailed outline of this two-phased placement methodology follows.

1. A mincut partitioning algorithm is used to generate a binary placement tree specifying the relative positions of the modules being placed.
2. A heuristic methodology is used to determine the exact rotations and mirrorings of each node of the placement tree as well as to establish the appropriate offset skew between the two children of each node.

7.2. Mincut Partitioning

The first step in developing the placement program was to select the general methodology by which placement decisions would be made. There were only a few such methodologies applicable to the general problem of placing rectangular modules. These included *mincut partitioning*, placement by *simulated annealing* [20], and a *knowledge based system* approach.

Although any of the methodologies mentioned above might successfully be used to generate reasonable placements of rectangular modules, the mincut methodology seemed the most likely to generate placements in a reasonable amount of computation time. Furthermore, the *slicing floorplans* generated by the mincut methodology define exactly the type of rough placement information that is required during the first phase of placement.

In general, mincut partitioning consists of the partitioning of a set of *elements* connected by a set of *nets* into two sets of elements such that the number of nets which are connected to elements in both sets is minimal. Some constraints on the relative sizes of the two sets must also be included otherwise it is clear that the optimal partitioning will result when all the elements are in one set and the other set is empty.

In this section, the application of a mincut partitioning methodology to the problem of module placement will be discussed.

7.2.1. The Mincut Heuristic

No polynomial-time algorithms are known to exist for solving the problem of mincut partitioning exactly. Because of this, a number of algorithms based on heuristics have been developed and applied to the problem [21] [22]. In practice, these algorithms tend to find reasonable solutions while running in near linear time.

This section describes the particular mincut heuristic selected for use in the placement program. It is similar in function and implementation to the mincut heuristic used by Koss in his topology generator program [7].

The mincut procedure takes as input a list of *elements* as well as a list of the *nets* connecting those elements. The set of elements is then partitioned such that the number of nets crossing the partition is small. The final partitioning of elements can also be constrained in a number of additional ways.

1. **Fixed Elements**

   Certain elements may be constrained to lie on one side of the partition or the other.

2. **Ordered Elements**

   Certain sets of elements may be ordered. Each element in the ordered set is assigned an index corresponding to its position in the ordering. The following must be true in order for a partitioning of the elements to be legal.

   Assume e and f are ordered elements.

   If index(e) < index(f),

   => if f is on the left, then e must be on the left.
   => if e is on the right, then f must be on the right.

   In the case of a vertical partitioning, left and right are replaced by top and bottom respectively in the above restriction.

3. **Independent Balance Constraints**

   The set of elements to be partitioned can also be divided into several smaller disjoint sets of elements. In order for a partitioning of the set of elements to be legal, each of the disjoint sets must independently satisfy a balance constraint on the relative sizes of the two halves of the partition.

7.2.1.1. Partition Balance Constraints

Partition balance constraints have only been discussed in general terms up to this point. In this section, some more formal definitions will be presented.

The basic idea behind balance constraints is to assure that the two partitioned halves generated by the mincut procedure are roughly equivalent in weight. The weight of each half is defined to be the sum of the weights of the elements in that half.

In applying the mincut partitioning methodology to the problem of placement, the weight of each element corresponds to the area of the associated module. Balance constraints in this case thus ensure that the two partitioned halves are roughly equivalent in terms of module area.

Desirable area balance characteristics have been obtained by constraining the element partitioning balance as follows.

- Let W_L be the sum of the weights of the elements on the left side of the partition.
- Let W_R be the sum of the weights of the elements on the right side of the partition.
- The partition is *balanced* if and only if all of the following relations are satisfied.

  1. $|W_L - W_R| < \Delta_{MAX}$
  2. $W_L > 0$
  3. $W_R > 0$

The first constraint ensures that the difference in areas between the two partitioned halves is below some threshold level. The second and third constraints assure that each half of the partition includes at least one module. This guarantees that the recursive application of the mincut procedure to a hierarchical placement problem will eventually terminate.

7.2.1.2. The Heuristic Mincut Procedure

Each invocation of the heuristic mincut procedure is iterative in nature. The procedure begins with an initial balanced partition. A sequence of passes is then executed with each pass reducing the number of nets crossing the partition.

Each pass consists of a sequence of steps in which elements are moved from one side of the partition to the other. At each step, the element to be moved is selected on the basis of which move will result in the highest *gain* where gain is defined as the reduction in the number of cross nets. It should be noted that the gain of the selected element may actually be negative. By allowing moves of modules with negative gains, the procedure is able to proceed beyond a locally optimal partition. After an element is moved, it is marked as *locked* and it is not moved for the remainder of the pass. The pass continues until no elements remain which can be legally moved. It is not legal to move an element if that element is locked or if the movement of that element will violate the partition balance.

The best partition found during the pass is saved and is used as the initial partition for the next pass. The process terminates when two successive passes result in no further improvement in the number of crossing nets.

The procedure runs in linear time per pass, and the number of passes tends to be very small even when a very large number of elements are involved. In practice, the complete algorithm thus runs in near linear time.

It should be noted that the mincut procedure used in the placement program is slightly different from the procedure presented here. More specifically, in the placement program version the *cost* of each partition is defined to be the sum of the net weights of each net crossing the partition rather than simply the number of crossing nets. This allows some nets to be given higher weights than others and allows net weighting information supplied by the user to be incorporated into the placement process.

7.2.2. Terminology

Before describing in more detail the application of mincut partitioning to the problem of placement, some terminology will be introduced for use in describing and defining a placement problem more exactly.

- **Box**

  A box is a data structure which contains all the information necessary to define a specific placement problem. A box has four sides (top, bottom, left, and right). Each side has associated with it a list of connections as well as possible constraints on the relative positioning of those connections. Each box also includes a list of the modules to be placed within that box.

- **Module**

  A module is a rectangular sublayout to be placed by the placement program. Each module corresponds to a node in the circuit description hierarchy. The node may correspond to a single leaf sublayout or to a larger fraction of the circuit consisting of several sublayouts. Each module has associated with it a set of terminals as well as the set of nets connected to those terminals.

- **Connection**

  A connection is a data structure used to represent the intersection of a net with the side of a box.

- **Net**

  A net is a data structure used to represent the electrical connectivity between terminals and/or connections.

- **Element**

  An element is a member of the set of objects to be partitioned using the mincut algorithm. An element is either a *connection* or a *module.*

7.2.3. Applying Mincut Partitioning to the Placement Problem

In this section, the exact means by which mincut partitioning is applied to the problem of placing modules will be discussed.

All the details needed to define a particular placement problem are included in a *box* data structure. This structure defines a set of modules to be placed as well as any constraints on where the external connections to the generated placement must be made. A box data structure is illustrated in figure 7-1.

Mincut partitioning can be applied to a box data structure (in either a vertical or a

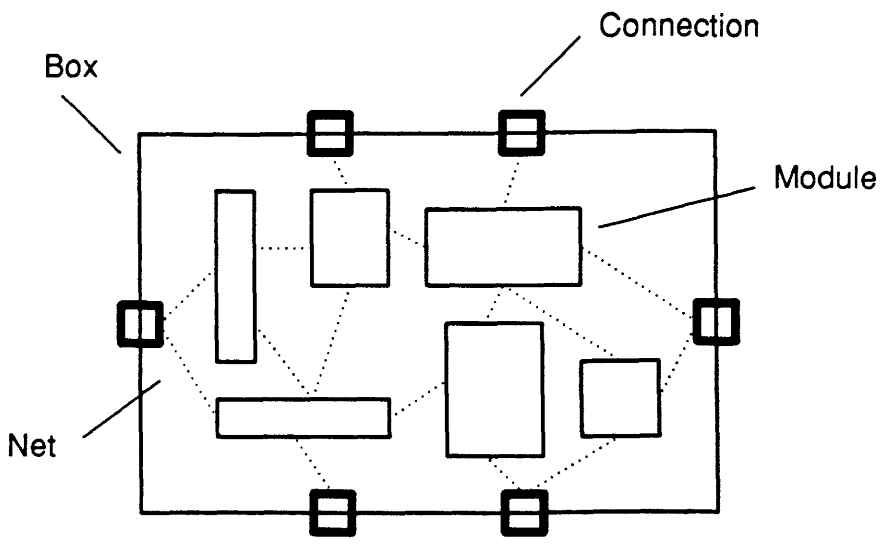

Figure 7-1: Box Data Structure

horizontal manner) to generate two smaller box data structures. The partitioning procedure can then be recursively applied to the newly generated box structures as well. By continuing this process until all of the boxes have been reduced such that each contains only a single module, the relative placements of all of the modules can be defined.

The following is a description of the application of mincut partitioning to a box data structure. The description assumes a partitioning of the box in which two smaller horizontally adjacent boxes are generated. A description of the vertical partitioning of a box would be nearly identical.

1. The elements of the box are divided into three disjoint sets. These sets are defined as follows.

    - **Top Elements** The top side connections. These elements may be constrained to be ordered if the top side connections were ordered.
    - **Middle Elements** This set includes all the modules included in the box as well as the left and right side connections. The left and right side connections are designated as fixed elements.
    - **Bottom Elements** The bottom side connections. These elements may be constrained to be ordered if the bottom side connections were ordered.

2. The mincut partitioning algorithm is then applied to the union of these three sets of elements with each set subject to an independent balance constraint.

3. Finally, the partitioned set of elements is used to generate two new box data structures. It should be noted that the cut set of nets for the partition of the original box is used to specify the right side connections of the new left box as well as the left side connections of the new right box.

The process of partitioning a box data structure is illustrated in figure 7-2.

7.2.4. Top-Down Mincut Partitioning

The incorporation of the mincut placement methodology into SOLO is now described. The mincut module placement procedure is applied in a top-down recursive manner to the restructured circuit description hierarchy. Each application of the mincut procedure adds two children to a binary placement tree that is simultaneously being constructed. The final binary placement tree describes a slicing floorplan of the circuit.

Initially, the binary placement tree consists of a single node and the mincut procedure is applied to the root of the circuit description hierarchy. A box data structure is defined which includes the children of the root as modules and the external circuit connections as connections. The area of each module is defined to be the sum of the areas of the leaf sublayouts that are descendents of that node in the hierarchy.

The mincut procedure is recursively applied to this box data structure generating smaller and smaller boxes until each box contains a single module. At the end of each mincut step, two more children are added to the appropriate leaf of the binary placement tree with each node corresponding to one subset of modules.

When a box which includes only a single module is generated, one of two operations are performed. If the module corresponds to a leaf sublayout (i. e. it is a leaf of the circuit description hierarchy), then that module is simply made a leaf of the placement tree. If the module corresponds to a node in the description hierarchy, then the box data structure is modified to include the children of that node as modules rather than the single parent module. The mincut partitioning process then continues on the new box data structure.

It is clear that each mincut partitioning step can be applied in either the vertical or the horizontal direction. If a desired aspect ratio has been specified, then the partitioning steps are constrained to be in the appropriate direction to a depth of

$$Depth = log_2 \frac{Max[W_{Desired}, H_{Desired}]}{Min[W_{Desired}, H_{Desired}]}.$$

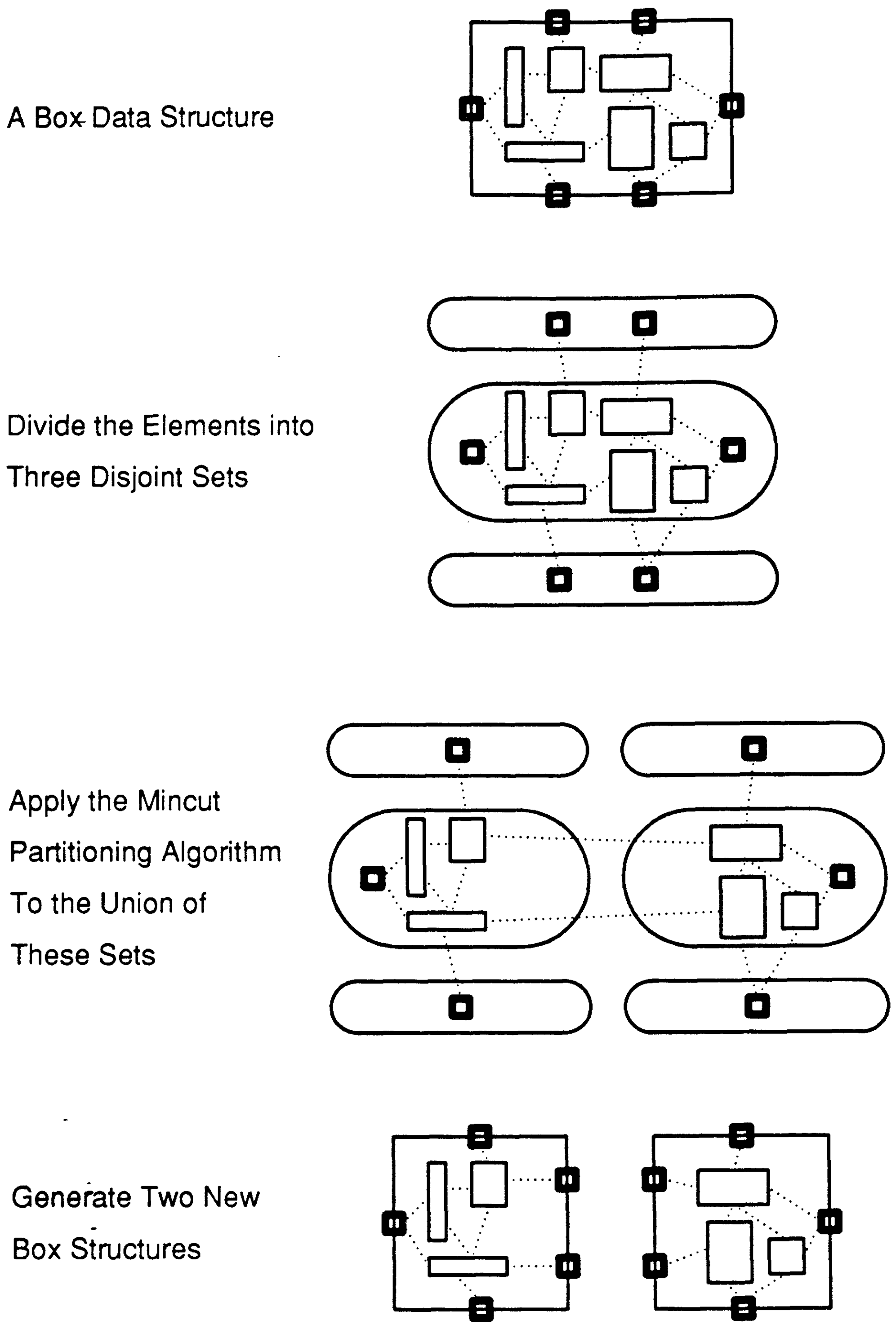

Figure 7-2: Partitioning A Box Data Structure

Beyond that depth (and at all depths when no desired aspect ratio is specified), both a horizontal and a vertical partitioning are tried and the one with the lowest partition cost[7] is selected.

7.2.5. Binary Placement Tree

The top-down application of the mincut procedure generates a binary placement tree specifying the relative positions of the leaf modules. A binary placement tree and the corresponding slicing floorplan are shown in figure 7-3.

In this section, a more detailed description of the binary placement tree data structure will be presented.

A binary placement tree is a tree data structure. Each node in the tree has a pointer to its parent as well as pointers to two children (except for the leaf nodes which obviously have no children). Each leaf node instead has a pointer to an associated module.

Each node of the tree also has a number of parameters associated with it which specify more exactly the relative placement of that node's children. More specifically, each node has associated with it a *partition orientation*, a *spacing*, a *skew*, and a *transform*.

The partition orientation associated with a node specifies whether the two subtrees pointed to by that node will be placed above and below each other (if the orientation is vertical) or adjacent to each other (if the orientation is horizontal).

The spacing parameter associated with a node specifies how much space will be left between the two children of that node. Similarly, the skew associated with a node specifies the relative offsets of the two children.

Finally, the transform associated with a node specifies how the placement specified by that subtree should be mirrored and or rotated before being incorporated into its parent node's placement.

The geometric meanings of the above terms are illustrated in figure 7-4.

[7]The partition cost is a simple function that takes into account the weighted net partition cost as well as the area balance of the partition.

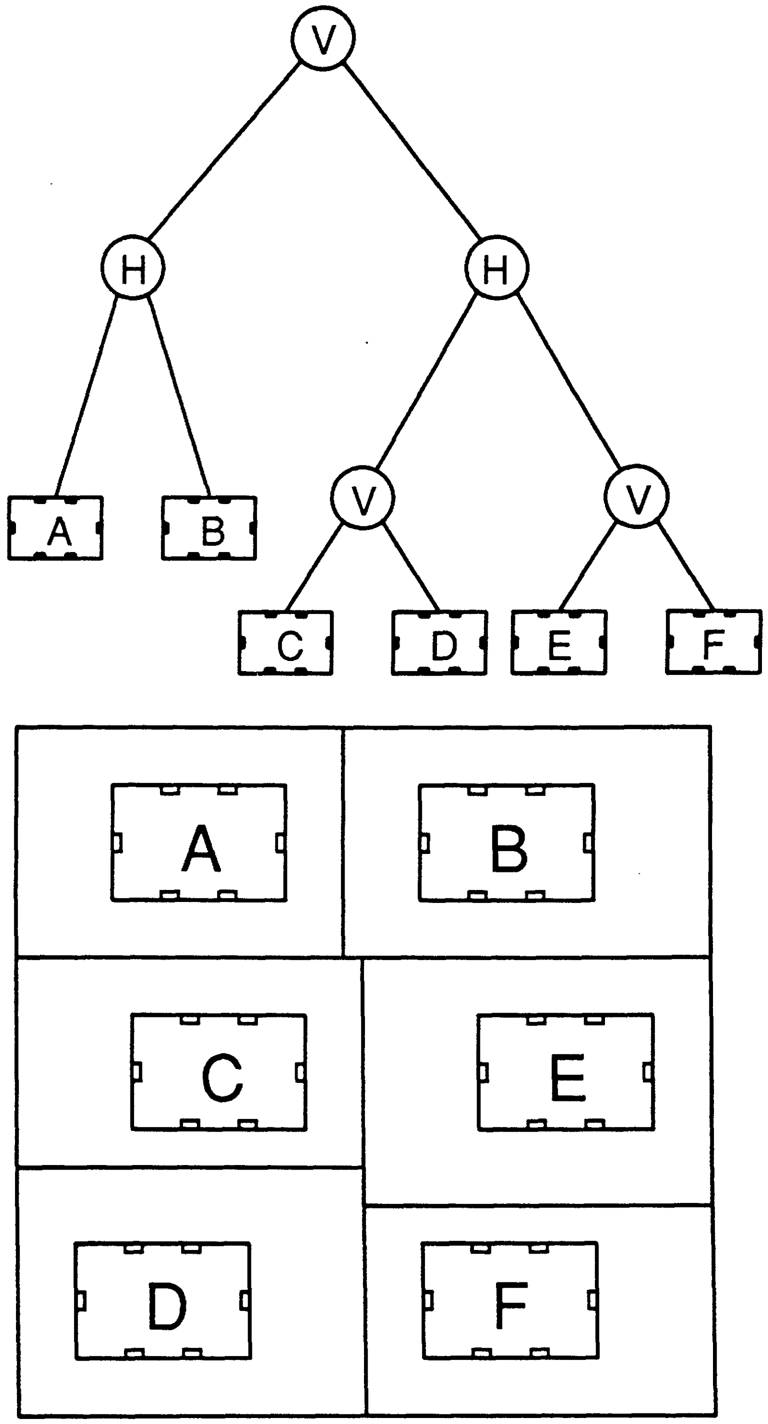

Figure 7-3: A Binary Placement Tree and the Corresponding Floorplan

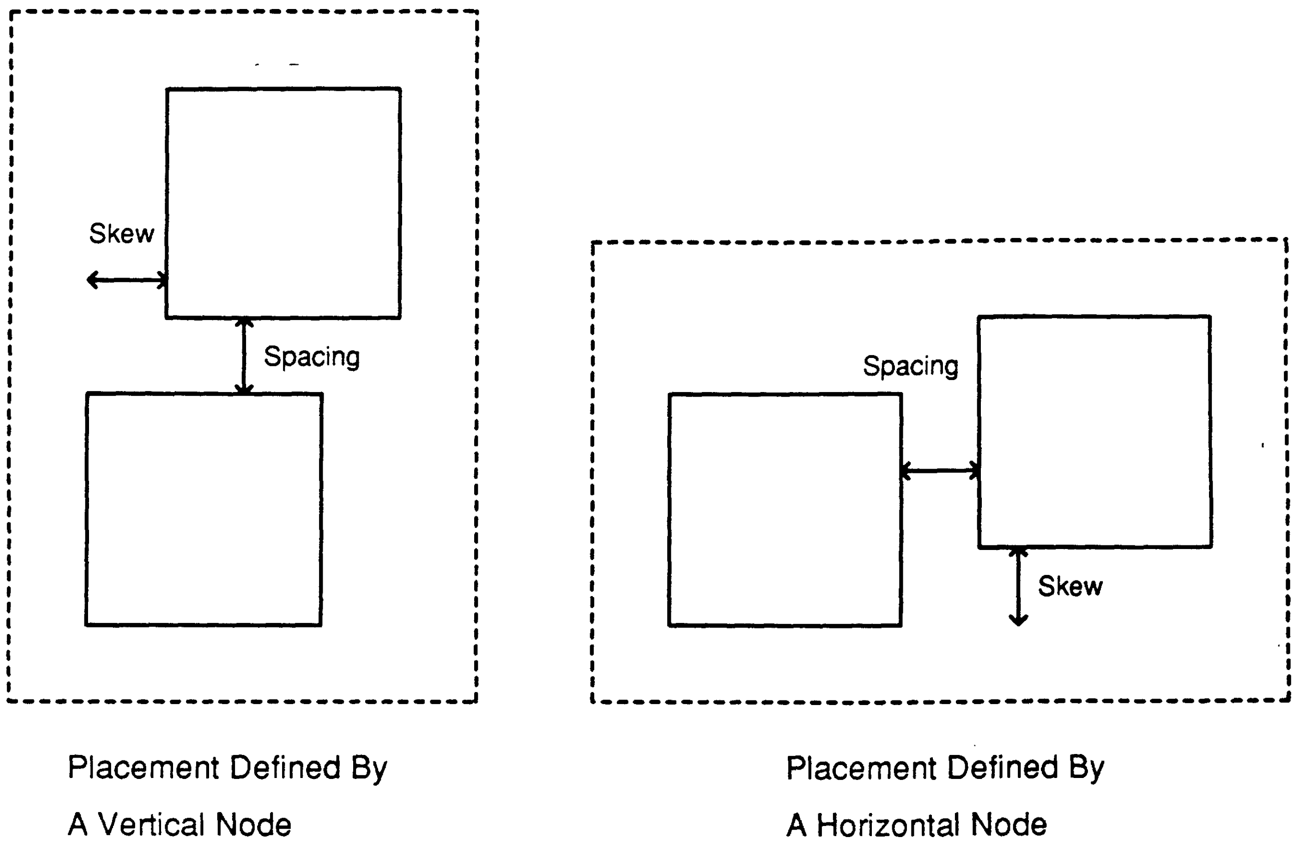

Figure 7-4: Relative Placement Parameters

7.3. Placement Refinement

Once the mincut partitioning algorithm has been used to generate a binary placement tree defining the relative positions of the modules being placed, a number of heuristic procedures are used to refine this rough placement.

The mincut partitioning establishes the form of the binary placement tree and associates a partition-orientation with each node. The transform, skew, and spacing parameters remain unbound. This refinement phase involves giving appropriate values to these parameters. The goal of this phase is to determine a transform, skew, and spacing for each node that will minimize the cost of the layout both in terms of area and electrical performance.

7.3.1. Optimization Techniques

Given a large number of parameters to be adjusted in order to minimize the cost of the associated layout, it was necessary to select an optimization methodology for determining the appropriate combination of parameter values. A number of possible techniques were considered and are listed below.

1. **Exhaustive Search**

   In order to guarantee an optimal solution, one can do an exhaustive search of all possible combinations of parameters. Each combination is scored and the combination of values yielding the highest score is selected. Such an approach is combinatorial in the number of modules however and thus it is not practical for use with placement trees of any size.

2. **Limited Search**

   Another approach is to visit each node of the tree only once, trying all possible parameter values for the given parameter for the visited node, and to select the value yielding the highest score. While this approach is certainly computationally efficient, it does not guarantee anything near an optimal solution. This technique ignores the fact that the environment surrounding each node being optimized may be changed by the subsequent optimization of other nodes.

3. **Probabilistic Optimization**

   Still another approach is to use a probabilistic optimization technique such as simulated annealing [20]. While such techniques tend to generate near optimal solutions, they also tend to be computationally expensive.

7.3.2. Transforms

The transform is the refinement parameter that has the most significant effect on the form of the associated placement. This parameter specifies how the subplacement associated with each node will be mirrored or transformed before it is incorporated into the placement as a whole.

In developing a methodology to select appropriate transforms for the nodes of the placement tree, one of the first questions to be addressed concerned exactly which node's transforms should be adjusted. Should only the transforms of leaf nodes be varied or should the transforms of all nodes be included? The PI system only adjusts the orientations of the leaf modules of the placement tree and thus it seemed that such a limited adjustment scheme might be appropriate. This was not the case however. Because of the low level layout issues involved in the placement of the small sublayouts typically generated by SOLO, an approach which adjusted the transforms of all nodes proved to be a necessity. A technique which adjusted the transforms of all the nodes in the placement tree was thus developed.

Next, it was necessary to develop a methodology for selecting the appropriate combination of node transforms. While computational considerations indicated that only a limited search technique could be employed, it was hoped that the technique could be augmented in some way so that the quality of the generated placements could be improved.

It was observed that a sublayout associated with a given node in the placement tree is most closely associated with the sublayout corresponding to the sibling node of the given node. This is to be expected given the nature of the mincut algorithm. It thus seemed that better placements could be generated by using a technique in which all possible combinations of transforms for each pair of sibling nodes were tried.

A modified version of the limited search selection technique was thus developed. In the modified technique, as each node is visited, the transforms of the two children of that node are adjusted rather than the transform of that node itself. All possible combinations of transforms for the two children nodes are tried. Although this approach increases the number of configurations to be scored at each step from eight (the number of orthogonal transforms) to 64, the increase in computation cost is more than offset by the improved quality of the generated layouts. Furthermore, the number of configurations to be scored is still linear in the number of modules.

An outline of the developed recursive procedure for optimizing the transforms of a placement tree is given below.

```
1. If the left child of the root of the tree is not a leaf node
   then optimize the transforms of the left subtree.
2. If the right child of the root of the tree is not a leaf node
   then optimize the transforms of the right subtree.
3.
      a. Try all 64 possible combinations of transforms for the two children
         nodes of the root of the tree.
      b. Score the placement resulting from each combination of transforms
         and select the transform pair yielding the highest score.
```

c. Set the transforms of the two children to the values indicated by the selected transform pair.

The procedure described above is applied as shown to those nodes whose partition orientations were not fixed during the mincut process. In the case where one (or both) of the children nodes have partition orientations that were fixed due to aspect ratio considerations, the tried rotating transforms for that node are limited to those *even* transforms that will not change the fundamental vertical or horizontal nature of the partition orientation.

7.3.3. Spacing and Skew

The two remaining parameters that must be adjusted during the placement refinement phase are the skew and spacing associated with each node.

The skew parameter specifies the offset between the two subplacements associated with the two children of the relevant node. Selecting an appropriate skew for each node involves tradeoffs between area and electrical performance. This is illustrated in figure 7-5.

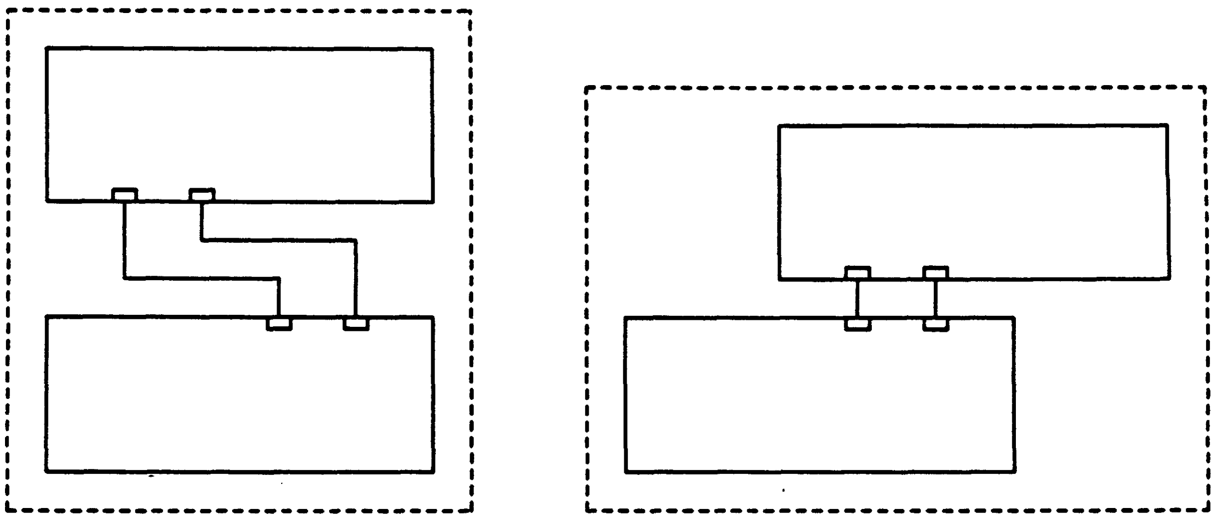

Figure 7-5: Tradeoffs between Area and Electrical Performance

The selection of a set of appropriate skews can be done using a limited search technique similar to that which was employed for transforms. In this case however, as each node is visited, a number of possible skews are tried and scored. The set of skews to be tried is limited to those for which the two subplacements being adjusted have at least some overlap. The minimum and maximum tried skews for a sample optimization problem are shown in figure 7-6.

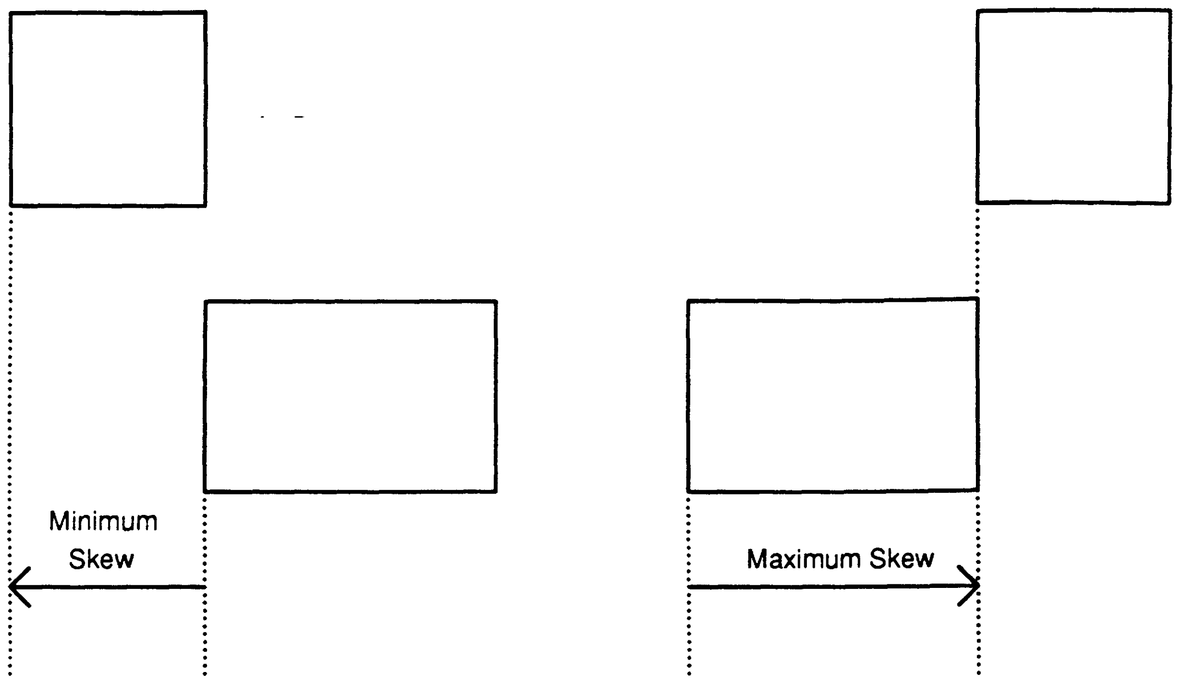

Figure 7-6: Minimum and Maximum Tried Skews During Skew Optimization

The problem of spacing adjustment is slightly different from the parameter optimization problems that have been discussed thus far. While the spacing between to subplacements should clearly be selected so as to optimize the performance of the placement as a whole, this spacing should also accurately reflect the space that will be required for routing.

A methodology for selecting appropriate spacing parameters must therefore have some means for estimating the amount of wiring space required between the relevant subplacements. Since global routing is not performed until after the placement phase has completed, this wiring estimation can be very difficult. Furthermore, the computationally expensive nature of global routing indicated that it would not be worthwhile to perform a preliminary global routing during the placement refinement stage.

A number of heuristic methodologies were developed and tested in an effort to provide an accurate means of estimating wiring requirements without the need to perform a global routing. None of these methodologies proved to be very accurate and at the same time some of them were very computationally expensive. Given the failure of even very complex routing estimation heuristics, the decision was made to employ a simple and computationally efficient routing estimator.

The layouts generated by SOLO have routing channels that while varying significantly in size tend to have somewhat similar aspect ratios. In other words, the width of a given routing channel tends to be a fairly fixed proportion of that channel's length. This observation led to the development of a function for estimating the appropriate spacing between two subplacements based only on the skew between, and relevant dimensions of, the two subplacements. Although this method of channel width estimation was not in an absolute sense very accurate, it performed as well as any of the much more complicated estimators and it was computationally very efficient. The ease with which the width of a routing channel could be changed during the subsequent routing stage (due to the nature of the binary placement tree) further indicated that such a rough estimator would be appropriate.

Thus the problems of optimizing the skew and spacing parameters were combined into a single optimization problem in which the skew is independently set and in which the spacing is expressed as a function of the selected skew.

An outline of the skew and spacing refinement procedure is given below.

1. **If** the left child of the root of the tree is **not** a leaf node
   then optimize the skew and spacing of the left subtree.
2. **If** the right child of the root of the tree is **not** a leaf node
   then optimize the skew and spacing of the right subtree.
3.
   a. Try all overlapping skews for the root of the tree starting with the minimum overlapping skew and continuing up to the maximum overlapping skew.
   b. For each skew, compute the appropriate spacing as a function of that skew.
   c. Score the placement resulting from each skew (and associated spacing) and select the skew and spacing yielding the highest score.
   d. Set the skew and spacing of the node to the values indicated by the selected skew and spacing.

7.3.4. Repeated Refinement

As was stated before, the limited search optimization technique selected for use in both refinement procedures presented here can find far from optimal solutions because it fails to take into account subsequent changes to other parameters when selecting the appropriate value for a given parameter. It was thus hypothesized that better results might be obtained by repeating the refinement procedures several times.

It was observed that the selected parameter values would converge on a stable set of values after roughly three optimization passes. The number of passes required was fairly independent of the size of the placement tree. This was not surprising given the local nature of the layout features that tended to be optimized by these parameter adjustments.

The full refinement phase was thus designed to include three passes of both the transform optimization and skew and spacing optimization procedures. The applications of the different procedures were interleaved in order to maximize the effectiveness of the repeated refinement.

The full placement refinement procedure is outlined below.

- The following is repeated three times:

    1. The transform optimization procedure is applied to the placement tree.
    2. The skew and spacing optimization procedure is applied to the placement tree.

7.3.5. Placement Scoring

The concept of *scoring* placements in order to determine the relative desirability of various placement configurations has been referred to in general terms many times throughout the discussion of placement refinement techniques. In this section a more detailed explanation of the exact methods and metrics used in evaluating placement configurations will be presented.

The evaluation metrics that have typically been applied to high level placement problems (such as placing the components of an entire chip) have been based almost exclusively on the evaluation of global circuit characteristics such as area or total interconnection length [23]. When evaluating placements on the scale of those generated by SOLO, lower level layout issues must also be taken into consideration.

In evaluating the desirability of a given placement configuration there are three basic factors that are considered.

1. Electrical Performance
2. Area and Aspect Ratio
3. Suitability for Well Positioning

Detailed explanations of the metrics that are used to evaluate these different placement characteristics follow.

7.3.5.1. Electrical Performance

In order to optimize a circuit layout in terms of electrical performance, it is necessary to minimize the interconnection lengths between connected nodes. Long interconnection paths result in the high resistances and parasitic capacitances that are typically associated with poor circuit performance.

A number of evaluation metrics geared towards the minimization of total interconnection lengths have thus been developed. Since intermodule routing is not accomplished until after the placement phase however, these metrics have been based only on estimates of the total wire length required. The methods used for estimating interconnection length given a set of net terminal locations have included the calculation of the *net half perimeter*, the *complete graph*, the *steiner tree*, and the *minimum spanning tree* [23].

While these different techniques do provide varying levels of accuracy in estimating the interconnection length of the subsequent routing, each proves to be inadequate when applied to the low level placement problem presented here. The problem with these metrics stems from the fact that they fail to accurately model the true costs of interconnection that are incurred during the subsequent routing.

In the simplest terms, an interconnection path between two terminals can be characterized as one of two types, either a *short interconnection* or a *long interconnection*. These two types are described in more detail below.

1. **Short Interconnections**

   These interconnections involve terminals that are very closely situated. The associated interconnection wires typically involve few or no jogs and include no layer changes. They are sometimes effected by the simple abutment of the two relevant modules. The costs of these connections, both in terms of layout area and electrical performance are typically very low.

2. **Long Interconnections**

   These interconnections involve terminals that are separated by some distance. The associated interconnection wires generally involve many jogs and layer changes. In order to avoid conflicts with other modules, the interconnection paths often deviate significantly from the straight line path between the associated two terminals.

Metrics based simply on estimated total interconnection length necessarily weigh each unit of interconnection length with equal importance. A placement which results in a spanning tree made up of two interconnection paths each of length 50 will be scored the same as a placement which results in a tree with one path of length 5 and another of length 95.

A more appropriate metric would reflect the fact that a spanning tree made up of one long and one short interconnection is more desirable than one made up of two long interconnections. Although it might seem that such a biased metric might lead to placements with very long total interconnection lengths, such inefficient placements do not occur. The general form of the layout is fixed when the placement tree is initially constructed. The mincut partitioning algorithm ensures that this initial placement has associated interconnection lengths that are of a reasonable size. The subsequent optimizations effected during the placement refinement phase are very local in nature.

An interconnection length evaluation metric biased in favor of short interconnections was thus developed.

The general form of the evaluation procedure is outlined below.

1. A model of the interconnection paths that will constitute the subsequent routing is created.
2. A score is obtained for each of these paths based on its length. The scoring function is biased in favor of very short interconnection paths.
3. The scores for each of the interconnection paths are summed to provide a biased interconnection length metric.

The first step of the procedure generates a model of the interconnection paths that will be used in wiring the given net. Of the interconnection length estimators discussed here only the minimum spanning tree and steiner tree algorithms generate models of the exact paths that will used during routing. Although the steiner tree provides the most accurate model of the subsequent routing paths , the steiner tree problem is NP complete [24]. A steiner tree based estimator is thus not well suited to the computational requirements associated with a procedure that must be executed many times. Conversely, a minimum spanning tree can be calculated in an efficient manner while at the same time providing a reasonably accurate model of the subsequent wiring paths. A minimum spanning tree algorithm is thus used to provide the needed model of the interconnection paths that will be included in the subsequent routing.

Given a spanning tree which defines the exact form of the estimated wiring, a scoring function which can associate an appropriate score with each of the constituent interconnection paths is needed. The function should calculate a score as a function of interconnection distance. It should further be biased such that the total score associated with a spanning tree made up of one short and one long interconnection path will be higher than the total score of a spanning tree (of equal total length) made up of two medium length paths. A scoring function well suited to this task is given below. A plot of the function is shown in figure 7-7.

$$Score = 50(e^{-d/5} + e^{-d/20})$$

The distance units used in the scoring function represent grid units in the target symbolic layout environment. When mapped into mask level layout, each unit corresponds roughly to the distance between the center of a minimum width wire and the center of an adjacent contact at minimum design rule spacing. At a distance of zero units, the function attains its maximum value of 100.

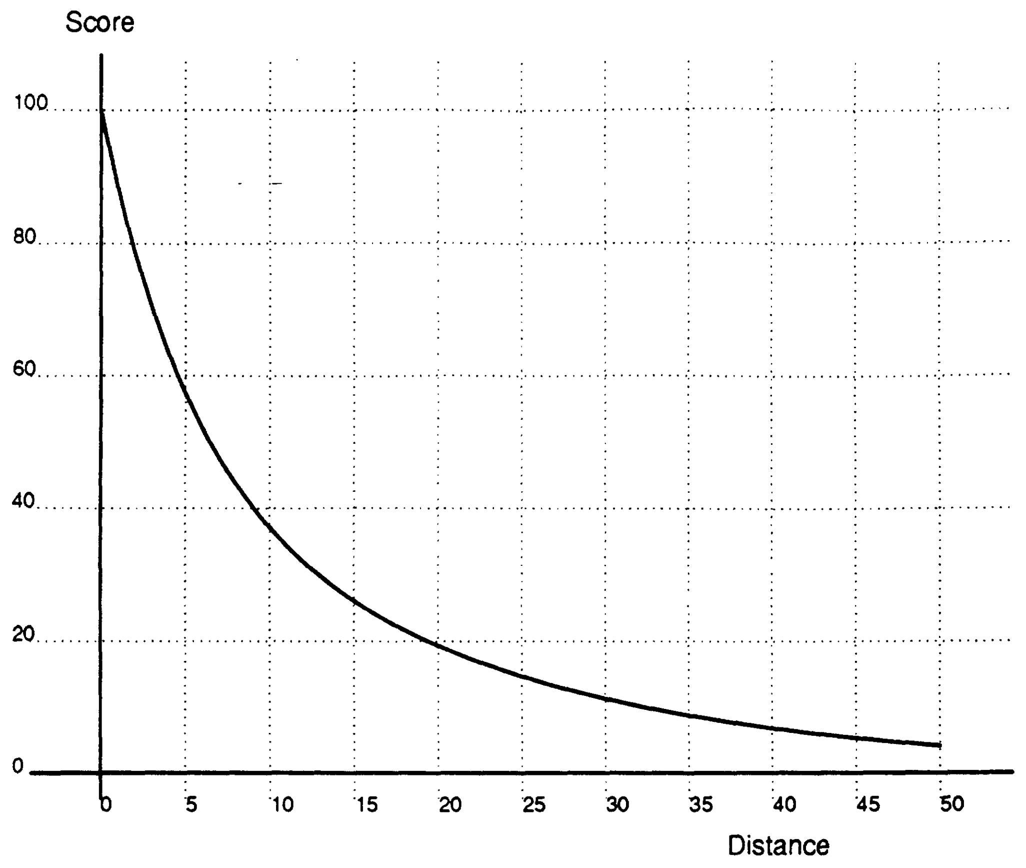

Figure 7-7: Interconnection Distance Scoring Function

The function has a number of desirable properties. Most importantly, it scores short interconnection distances more highly than long interconnection distances. Furthermore, the function is not linear. The exponential factors ensure that the slope of the function decreases as distance increases. This has the effect of scoring a movement which decreases an interconnection distance from 5 to 2 more favorably than one which decreases a distance from 55 to 52.

The scoring function includes two exponential factors, one with a time constant of 5 units and the other with a time constant of 20 units. The factor with the shorter time constant gives the expression very high gain at short distances. This gives decreases in interconnection lengths at very close distances high importance. Two connections that are closely situated will thus tend to be brought even closer together so that a wire jog may be avoided or a connection by abutment may be made. The factor with the longer time constant ensures that reasonable gain still exists at longer distances. An interconnection path of length 37 is scored more highly than a path of length 40.

This nonlinear scoring function was combined with a minimum cost spanning tree algorithm to provide an appropriately biased interconnection length metric. The procedure for calculating this metric is outlined below.

1. For each relevant net[8]:
    - a. Compute a minimum cost spanning tree.
    - b. For each edge (i. e. connection path) in the spanning tree:
        - i. Generate an associated score by applying the interconnection distance scoring function to the distance of that path.
        - ii. Scale this score by the relative weighting of the net.
    - c. Sum these weighted scores.
2. Sum the scores from each net.

It is interesting to consider how this biased interconnection length metric scores the two equal length spanning trees discussed earlier. Assume again that one has interconnection paths of lengths 50 and 50 units and the other has lengths of 95 and 5. The two trees are no longer scored the same. Applying the biased metric to the spanning tree with paths of lengths 50 and 50 results in a score of only 8. Applying it to the tree with paths of 95 and 5 units yields a score of 57.

7.3.5.2. Area and Aspect Ratio

Layout dimensions are the placement characteristics that are most easily captured in an evaluation metric. Each node has associated with it a width and height. These values are initialized immediately following the construction of the binary placement-tree and are updated appropriately with each change of a parameter during the refinement phase. The width and height values associated with each node in the tree are thus always kept accurate.

The estimated height and width of a placement tree can be obtained by simply using the height and width associated with the root of that tree. Although these values do not account for the space required to route to external connections and use only

[8]A relevant net is one for which the relative positions of its terminals may be changed by the current parameter adjustment.

estimates of the internal routing area, they do provide good rough estimates of the dimensions of the generated layout.

If all that was desired was a metric to measure the estimated layout area of a given placement configuration, such a metric could be obtained by simply multiplying the height and width values associated with the root of the placement tree. A metric which also measures the compatibility of a given placement configuration with the desired aspect ratio is needed however. Although the rough form of the placement tree is made compatible with the desired aspect ratio during the mincut partitioning phase, the introduction of aspect ratio considerations into the refinement phase tends to make the aspect ratios of the generated layouts match the desired aspect ratios somewhat more closely.

Aspect ratio considerations can be incorporated into a layout area metric by using an expression which scales each dimension of the placement tree by the desired size of the opposite dimension and then sums the square of these two terms. This placement tree *size* metric is given below.

$$Size = [W \cdot H_{Desired}]^2 + [H \cdot W_{Desired}]^2$$

For a given area (i. e. $W \cdot H = K$) , this expression is minimized when the height and the width are in the same ratio as the desired height and the desired width. This expression thus provides a metric that will be minimized when applied to placements that are small in area and are of the desired aspect ratio.

7.3.5.3. Suitability for Well Positioning

The basic idea in creating a placement that is well suited to well positioning is to cluster sublayouts that are of the same diffusion type. The spacing required between different types of diffusion tends to be quite large and thus an effort is made to avoid such interfaces. By grouping sublayouts that are of the same diffusion types the number of such interfaces can be minimized.

Originally, an evaluation metric was used in which the two different diffusion types were treated much like two signal nets each connected to the appropriate sublayouts. By trying to minimize the biased interconnection length metric associated with each of these *pseudo* nets, sublayouts which were of the same diffusion type tended to be clustered. Although this technique did work as desired, an unanticipated problem occurred.

While the metric did tend to cluster sublayouts of the same diffusion type, it made no effort to control the relative positions of the N-type and P-type diffusion areas. Recall that P-type diffusion areas generally have connections to Vdd while N-type areas generally have connections to Gnd. The positioning of Vdd and Gnd connections thus tended to be dominated by clustering considerations. Because the subsequent routing program was biased in the way it would connect to power terminals (only connecting to Gnd on two sides of a sublayout and to Vdd on the remaining two sides), the placements that were generated (primarily on the basis of clustering considerations) would often have connection paths to Vdd and Gnd that were unnecessarily long and indirect. Although one might have thought that the net interconnection length minimization metric would have helped to avoid such problems, this was not the case. This metric did not take into account the biased nature of the routing program when wiring power and ground.

The associated long meandering power lines introduced significant costs in the way of layout area. These costs were typically higher than those that would have occurred had the placement been guided by a *suitability for power wiring* metric rather than by clustering considerations.

It was thus necessary to develop an evaluation metric that would tend to cluster sublayouts of the same diffusion type (as before) but which would also ensure that placements in which the relative locations of N-type diffusion and P-type diffusion areas were well suited to the biased routing program would be scored more highly than those in which they were not.

Such a scheme was developed and is described below. The basis of this technique involves the association of a simple *diffusion type* with each node of the placement tree.

There are seven basic diffusion types one of which is associated with each node. The different types of nodes are listed below and are illustrated graphically in figure 7-8.

1. N type node.
2. P type node.
3. N-P type nodes (four different orientations).
4. Mixed type node.

Each diffusion type is meant to reflect the diffusion area characteristics of the

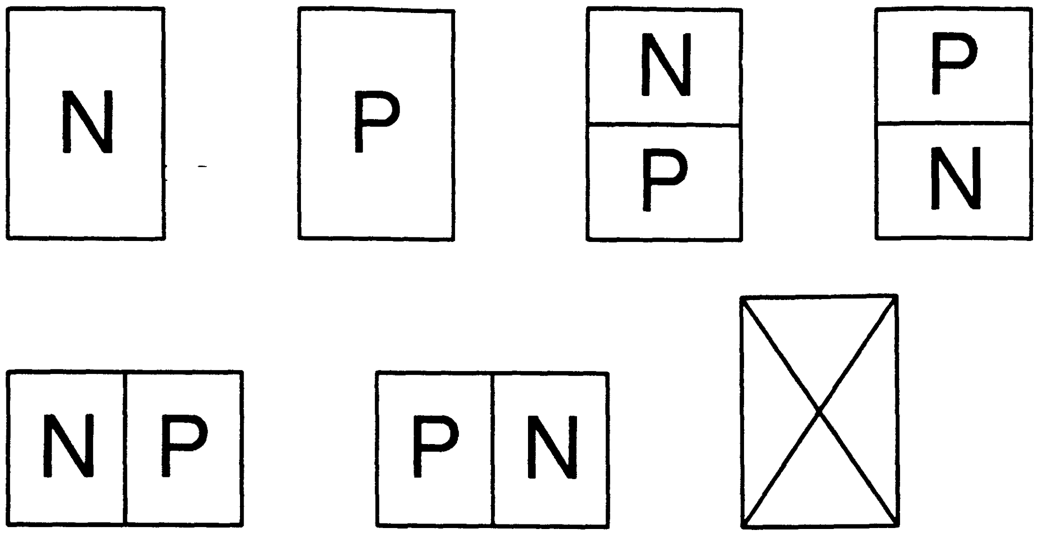

Figure 7-8: Graphic Representations of the Seven Different Diffusion Types

associated subtree. Leaf nodes are either N-type or P-type depending on the diffusion type of the associated sublayout. The types of the higher level nodes are determined by combining the types of the two associated children nodes in obvious ways. A number of typical node combining rules are shown in figure 7-9.

A binary placement tree in which the diffusion types of the nodes are graphically labeled is shown in figure 7-10.

Immediately following the construction of the binary placement tree, a depth-first procedure is used to give appropriate initial diffusion type values to each of the nodes in the tree. Because changing the transform associated with a node in the placement tree can effect the diffusion types of other nodes, a special procedure must be included to update the appropriate diffusion types whenever a transform parameter is changed. In this way the diffusion types associated with each node in the tree are always kept up to date.

Each of the seven diffusion types has an associated score based on its relative desirability. N, P and mixed diffusion types each have an associated neutral score since rotating a node of one of these types will not affect the basic nature of the placement in terms of N and P diffusion area positioning. The two N-P diffusion types which are oriented so as to be well suited to the biased routing of power wires are given high scores because power connections to the associated sublayouts can be made in an efficient manner. Conversely, the two other N-P diffusion types are given low scores. Making power connections to the associated sublayouts can be very expensive.

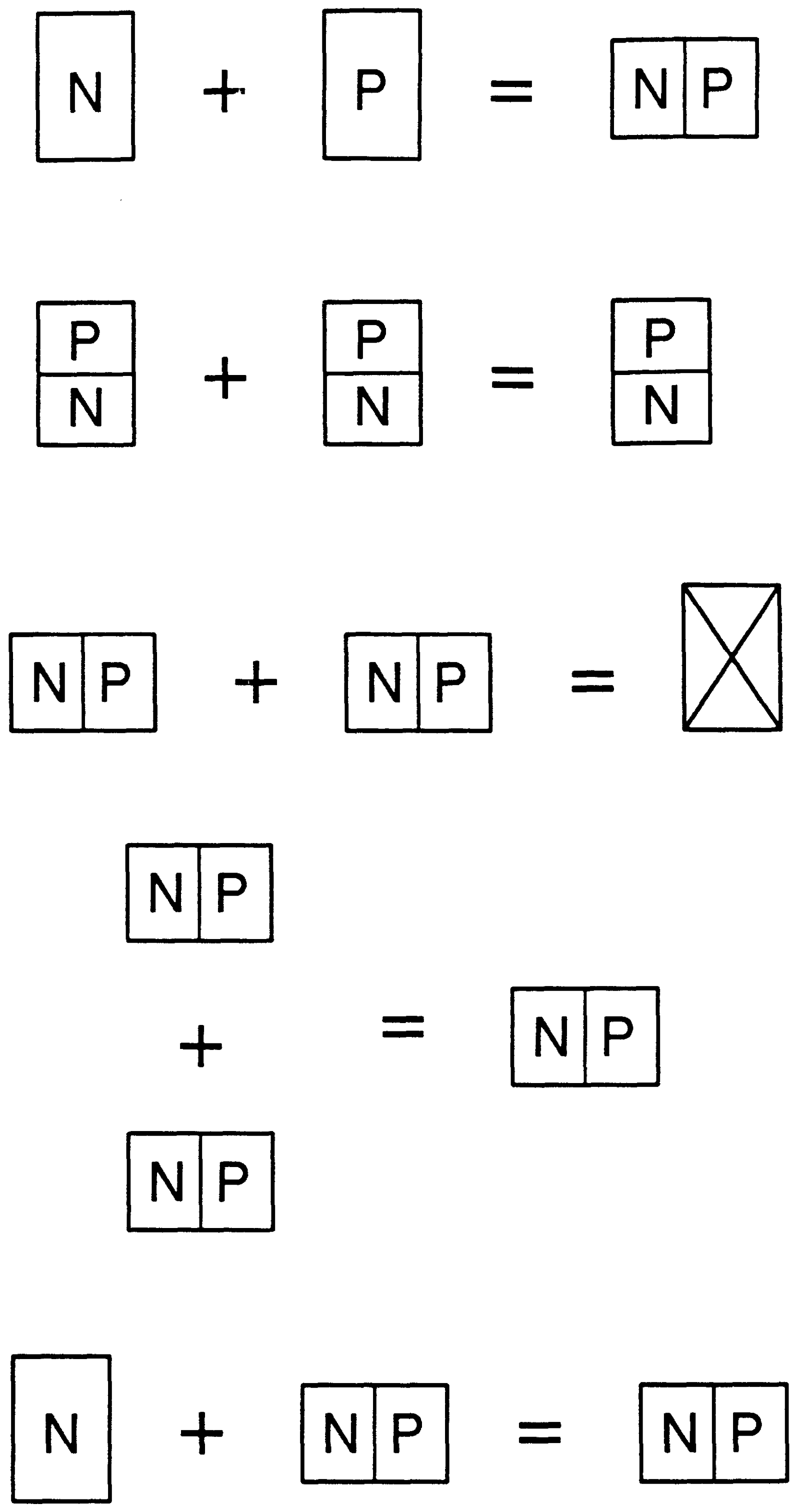

Figure 7-9: Some Typical Diffusion Type Combining Rules

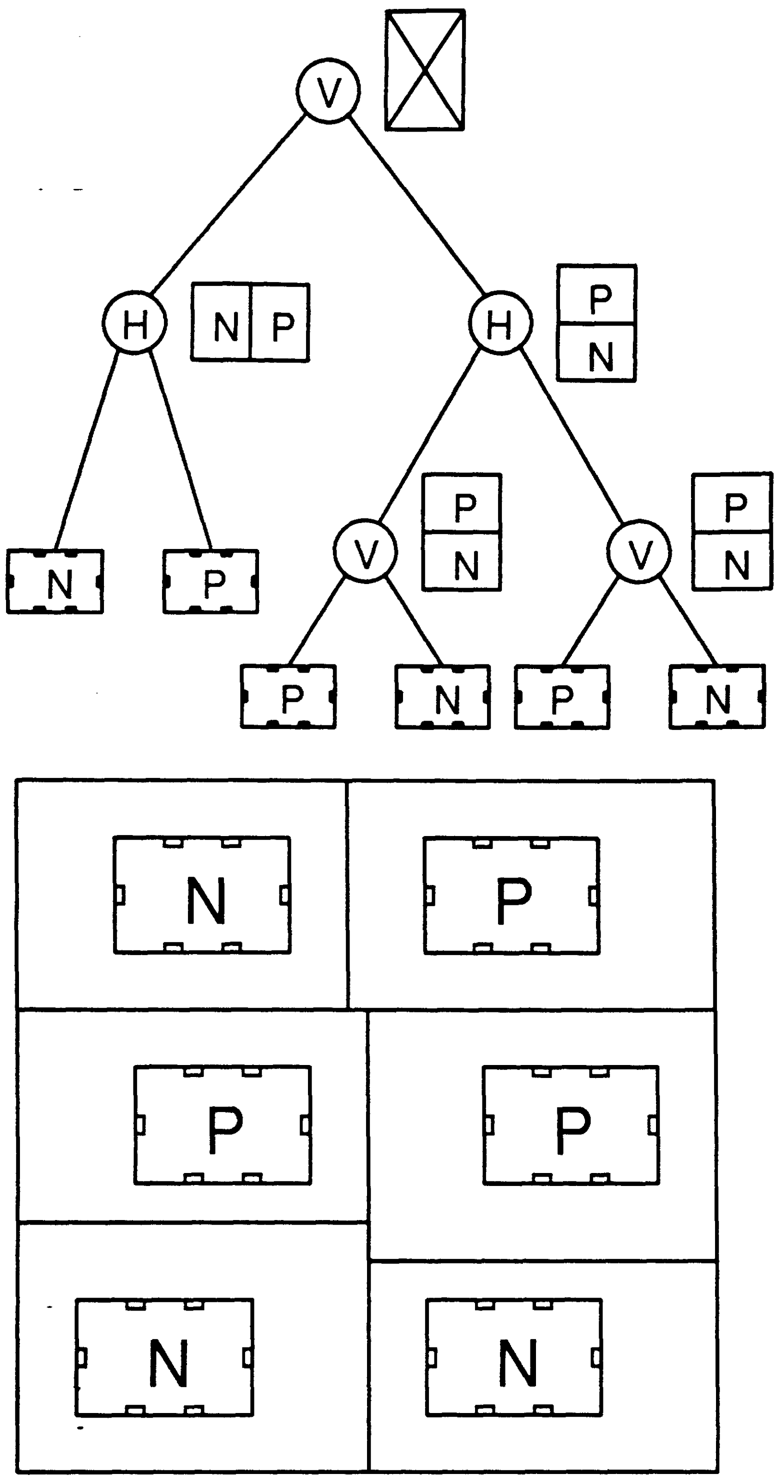

Figure 7-10: A Placement Tree and Floorplan Showing the Node Diffusion Types

In order to evaluate the overall desirability of a placement tree (in terms of suitability for well positioning), the scores associated with the diffusion types of each node in the tree are first scaled by the total area of the associated subtree. These weighted scores are then summed and a metric reflecting the relative desirability of the entire placement tree is obtained. A placement which is well suited to the biased routing of power nets will have many N-P type nodes oriented in the correct way and which thus have high scores. Similarly, a placement in which sublayouts of the same diffusion type are well clustered will have many high scoring N-P type nodes. The clustering will lead to high scoring diffusion types many levels up from the leaf level. A more fragmented positioning will instead lead to mixed diffusion types very close to the leaf level.

Chapter 8
Routing

After the placement of the leaf modules is completed, a routing program is used to define the exact placement of the wires interconnecting the various modules and the external connections. A set of specifications defining more completely the requirements of the routing program is given below.

- The program need not be totally general in nature. More specifically, it is not necessary to have the capability to define a set channels linking arbitrarily placed rectangular modules. The nature of the placement tree definition of the positions of the leaf modules is such that the appropriate routing channel structure is inherently defined. Furthermore, the defined channel structure will always be such that it is possible to route all the channels using channel routing rather than switch-box routing capabilities.
- The routing program must be able to effectively utilize information about terminals internally connected within a module. By using this information, redundant connections to internally connected terminals can be avoided.
- The program should be capable of routing both signal and power nets. The program should further ensure that power nets run exclusively in metal.

Although this set of specifications indicated that a direct reimplementation of any one of several existing routing programs would produce acceptable results, a survey of the techniques employed by these existing programs led to the conclusion that better results could be obtained by using a slightly different routing technique specifically designed for the intended symbolic layout routing environment. The fact that the requirements of the routing program were in some ways less stringent than those for a truly general routing program (due to the nature of the placement hierarchy generated by the placement program) also indicated that an application specific routing program would be appropriate.

Thus, a new placement program was written for use as part of SOLO. A detailed description of this routing program follows.

8.1. Overview

In this section a general overview of the routing problem is given and an outline of the channel routing approach to solving that problem is presented.

8.1.1. The Routing Problem

The complexity of the routing problem is such that it is impossible to solve the problem optimally for instances containing more than a few modules and a few nets. As a result, the algorithms generally used to solve the routing problem have been heuristic in nature and have imposed a number of simplifying assumptions on the form of the completed routing [25].

There are a number of different heuristic methodologies that are typically applied to the routing problem.

- *maze-running routers* [26]
- *line probe routers* [25]
- *channel routers* [27]

While each of these methods has a number of associated advantages and disadvantages, only the channel routing methodology can provide a reasonable guarantee of routing success by considering all nets simultaneously during the routing process. Furthermore, channel routing is among the most computationally efficient of the above methodologies. The channel routing methodology was thus selected as the basis of the routing program.

8.1.2. Channel Routing

Channel routing simplifies the routing problem by partitioning the problem into a number of simpler subproblems.

1. First,the routing surface is divided into a number of disjoint rectangular areas called *channels*. The routing surface is that part of the layout that is not occupied by the placed modules.
2. Next, a *global router* selects which channels each net traverses.
3. Finally, a *channel router* completes the detailed routing of each channel. This final routing is generally constrained to utilize different layers for different directions. All horizontal interconnection segments are routed in one layer while all vertical segments are routed in a second layer that is insulated from the first. This requirement is relaxed in the case of power nets.

8.1.3. Terminology

In this section the terminology used in specifying a routing problem is defined.

- **Module**

  A module is an atomic, rectangular sublayout that has been placed by the placement program. Each module has associated with it a set of nodes and their associated terminals.

- **Terminal**

  A terminal is a point where a connection to a node can be made. Each terminal has an associated position and layer. It is assumed for the purposes of routing however that any terminal can be accessed on either of the two routing layers by the addition of a single contact (if necessary). While terminals are generally located on the borders of modules, a terminal may also be used to represent a connection point at the end of a channel or a point on the border of the layout where an external connection is to be made.

- **Node**

  A node is a set of terminals which are electrically connected prior to routing. All those terminals which are electrically connected within a given module are associated with a single node.

- **Net**

  A net is a data structure used to represent a set of nodes to be interconnected by the routing program.

Modules and terminals are illustrated in figure 8-1.

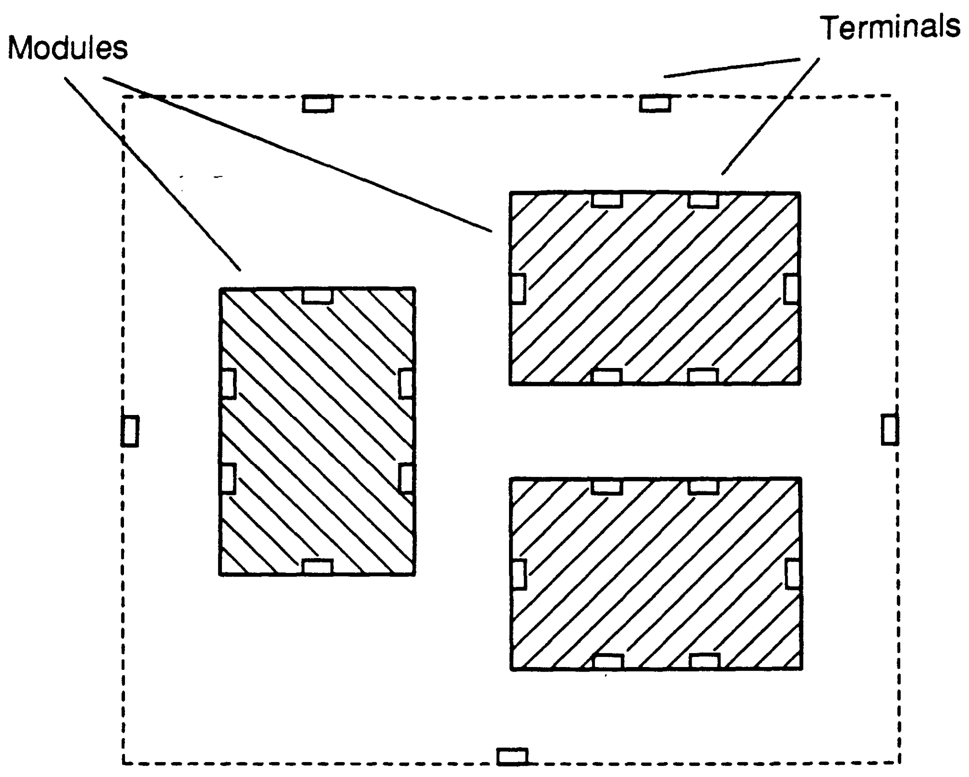

Figure 8-1: Routing Problem Terminology

8.1.4. The Routing Process

The input to the routing program consists of set of placed modules (each with associated nodes and terminals) as well as a number of external connections as represented by node/terminal pairs.

The goal of the routing program is to interconnect all the nodes associated with each net. Connections to a node can only be made through an associated terminal. If a node has more than one associated terminal, whichever terminal is most convenient may be used. A signal net may even be routed through a module by entering a node through one terminal and exiting through another.

The routing program accomplishes this interconnection through the application of a standard channel routing methodology. A summary of the complete routing process is given below.

1. A routing channel is created for each node of the placement tree. Each channel corresponds to the rectangular routing area between the two children of the associated node.
2. Four surrounding channels are added to provide a routing interface to the external connections.
3. Each net is globally routed, determining which terminals it will be connected to and which channels it will use.
4. The routing specification for each channel in the layout is generated.
5. A detailed routing of each of the channels is performed.

8.2. Channel Creation

The first step in the routing process is to define the dimensions and locations of the routing channels which make up the routing surface.

In the case of a placement of rectangular modules defined by a binary placement tree, the channel definition process is quite straightforward. A routing channel is associated with each node of the placement tree. The main dimension of the channel runs perpendicular to the orientation of the node. The width of the channel is defined by the spacing between the children of the node. After the channel has been placed, the effective sizes of the two leaf modules are expanded to extend to the ends of the routing channel. This composite module, consisting of two expanded modules and a routing channel will constitute a single module at the next level of the placement tree. Although expanding the two leaf modules in the manner described here prevents the expansion space from being used for routing, this operation assures that the composite module is rectangular. The area minimization metrics used during the placement phase assure that the area wasted due to this expansion is minimized. The structure of a composite module is shown in figure 8-2.

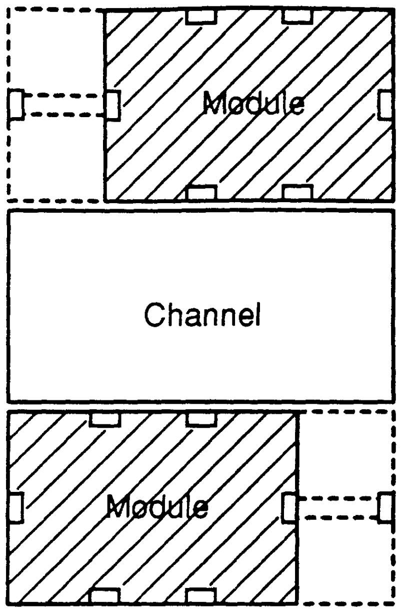

Figure 8-2: A Composite Module

This process of adding routing channels to the placement tree starts at the leaf nodes of the tree and continues to the root node. The set of channels associated with a complete placement tree is illustrated in figure 8-3.

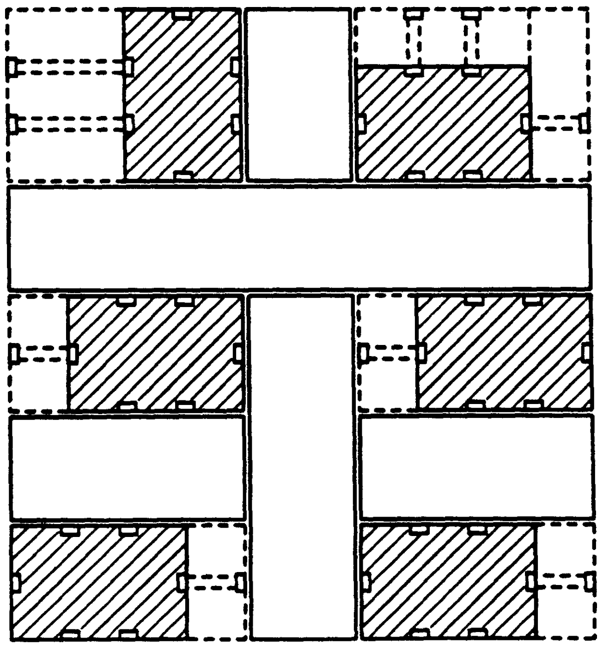

Figure 8-3: Routing Channels Associated with a Placement Tree

The set of channels defined by a placement tree provides the necessary routing surface to connect the terminals on different leaf modules but it provides no means for connecting those terminals to external connections. Four surrounding routing channels are thus added to provide an interface to the external connections of the layout. Although all four surrounding channels will rarely be needed, a complete set is included to accommodate all possible interconnection situations. Since unused surrounding channels will not contribute to the final layout area, it is not necessary to determine which surrounding channels are needed and which are not. A complete set of routing channels for a set of placed modules is shown in figure 8-4.

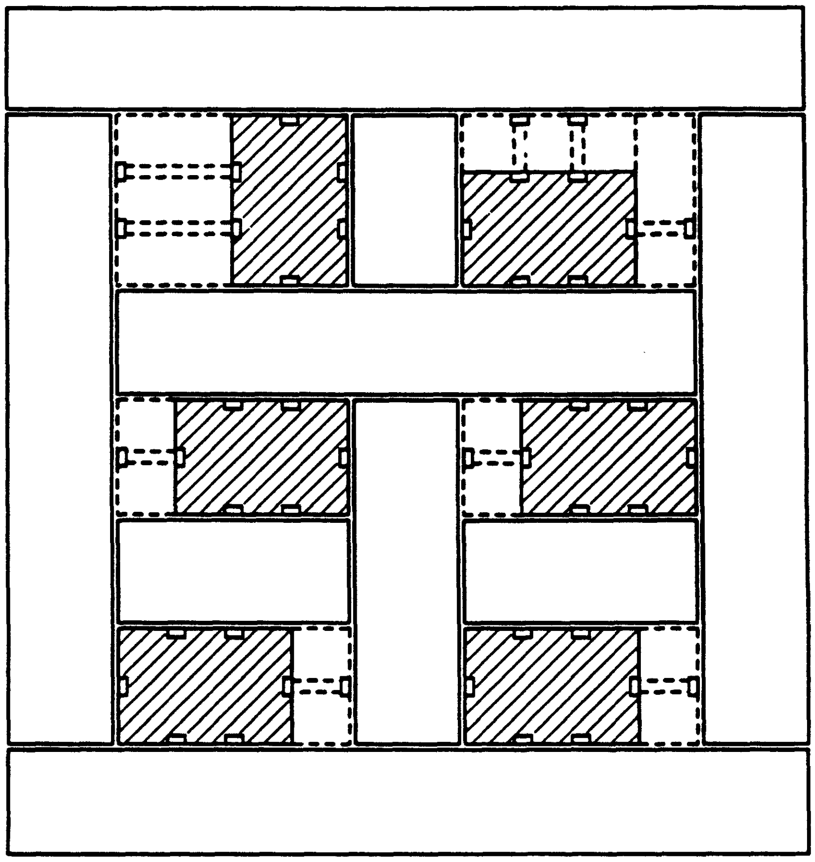

Figure 8-4: A Complete Set of Routing Channels

8.3. Global Routing

Once the routing surface has been defined, a global router is applied to each of the nets in the layout. The router determines which terminals each net will be connected to and which channels each net will traverse.

8.3.1. Generating a Graph Model

A graph model representing the relative distance between the terminals and external connections must be generated before the global routing can be accomplished. This graph model is generated as follows.

1. At each end of each routing channel a set of nodes and associated terminals is created. Each set contains one node/terminal pair for each net in the layout. These *pseudo* terminals are positioned at the center of the line defining that end of the channel.
2. Vertices are created and associated with each node in the layout.
3. Each terminal in the layout is associated with the appropriate adjacent channel or channels. Terminals on modules will typically be associated with a single channel while the terminals positioned at the ends of channels will be associated with the two adjacent channels.
4. For each channel:
    - For each signal net:

      All the terminals of that net associated with that channel are interconnected using a fully connected graph of edges. These created edges are added to a list including all the edges positioned within that channel.

      While in actuality it is the vertices associated with the terminals that are joined by edges rather than the terminals themselves, a reference to the connection of terminals (rather than to the connection of vertices) is made here. This is intended to stress the fact that an edge is added for every possible pair of terminals, not just for every possible vertex pair.
    - For each power net:

      All the *usable* terminals of that net associated with that channel are interconnected using a fully connected graph of edges. These created edges are added to a list including all the edges positioned within that channel.

      For the Vdd net, usable terminals are those terminals associated with the bottom and right of the channel (if the channel is horizontal) or those terminals associated with the top and left of the channel (if the channel is vertical).

For the Gnd net, usable terminals are those terminals associated with the top and left of the channel (if the channel is horizontal) or those terminals associated with the bottom and right of the channel (if the channel is vertical).

These restrictions are relaxed somewhat on two of the external channels to allow feedthrough power busses to be formed.

By restricting the usable terminals for power nets in this manner, it can be assured that no power net will ever cross a channel (by being connected to both the top and bottom of a horizontal channel or to both the left and right of a vertical channel). It is further assured that the Vdd and Gnd nets will never cross. This allows power net wiring to be accomplished on a single layer.

5. Finally, a weight is assigned to each edge of the graph. The weight associated with each edge is a complex function based on the manhattan distance between the two terminals associated with that edge as well as a number of additional penalty functions.

   One penalty function is applied if the two terminals associated with the edge are on different layers. This additional weighting gives preference to edges connecting terminals on the same layer over edges connecting terminals separated by a similar distance but connected to two different layers. An additional penalty is applied if the edge being weighted crosses an edge connecting the output nodes of a pull-up/pull-down pair. This penalty weighting is intended to allow the connecting wire between a pull-up and pull-down to be run in metal (or diffusion) whenever possible.

The routing graph associated with a typical signal net is illustrated in figure 8-5. The graph associated with a power net is shown in figure 8-6. Each of the contiguous darkened areas represents a vertex. Those that are small circles correspond to nodes with single associated terminals. The more unusual shapes correspond to nodes with multiple internally connected terminals. The rounded ends of these shapes are positioned at the associated terminal locations.

The diagram of power net routing illustrates the fact that connections to power terminals are only made on certain sides of modules. It is thus necessary to consider the constraints that must be placed on power terminal locations to ensure that any required interconnections can be accomplished. Only one such constraint need be applied.

- Each power node must be accessible through terminals on two opposite sides of the module.

If this constraint is followed, then any power node can always be wired regardless of the orientation of the associated module.

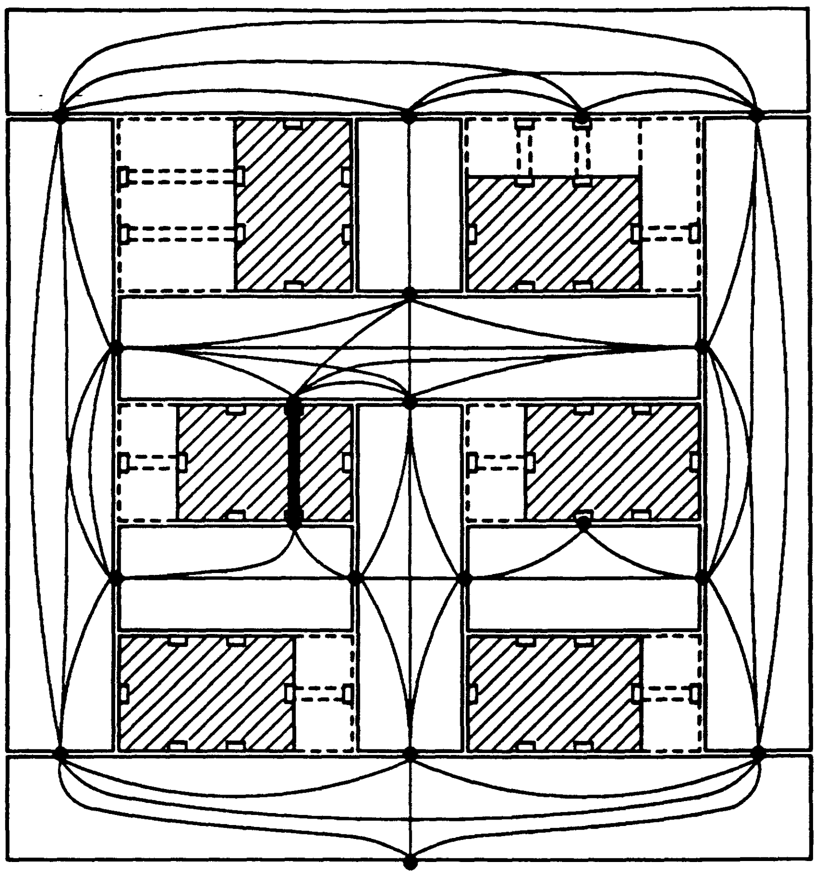

Figure 8-5: The Routing Graph Associated with a Signal Net

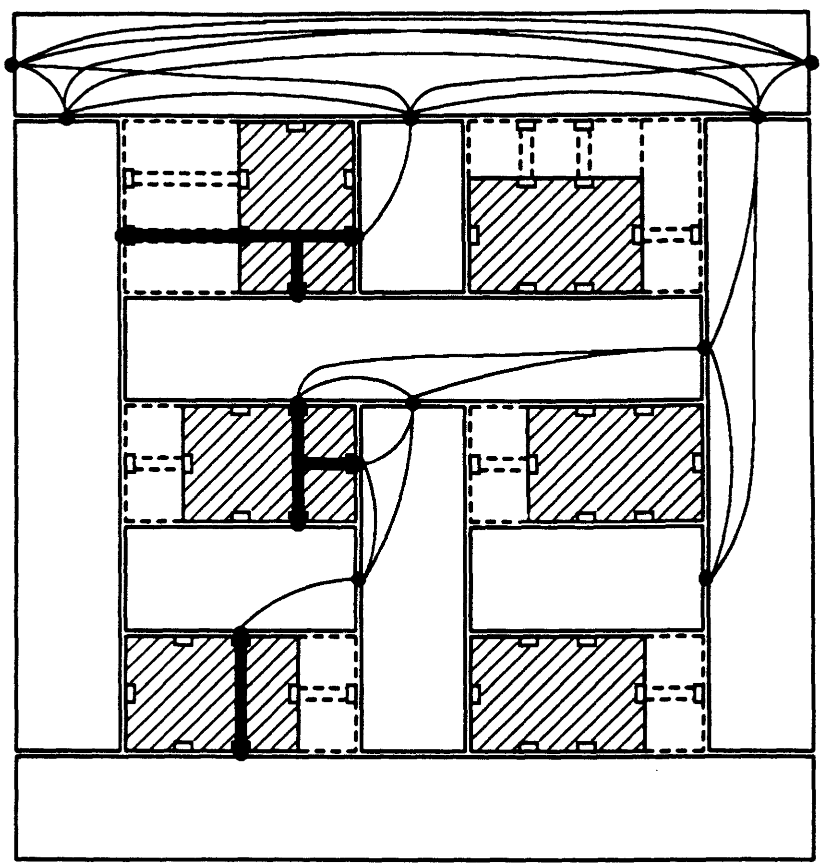

Figure 8-6: The Routing Graph Associated with a Power Net

8.3.2. Finding the Best Routing

Given a connected routing graph for each net in the layout, the problem of globally routing each net is reduced to the problem of finding an appropriate spanning tree of the associated routing graph.

It is important to notice at this point that the problem of globally routing the nets of the layout is not the same as the problem of finding the minimum weight spanning tree of the routing graph as it has been generated. The routing graph includes the pseudo vertices located at the ends of the routing channels. These vertices are intended to provide paths whereby the vertices associated with true terminals and external connections may be connected. It is certainly not desirable to have the global routing of each net connect each of the pseudo vertices associated with that net (one at the end of each routing channel). Instead, the global routing should define a spanning tree which includes all the true vertices of the graph and only those pseudo vertices which are needed to allow all the true vertices to be connected. Such a spanning is called a *Steiner Tree.* [24]

8.3.2.1. The Steiner Tree Problem

Before formally defining the Steiner tree problem, some formal definitions of weighted graphs and spanning trees will be presented. The definitions given below are taken from [28].

- An undirected weighted graph $G=(V,E,w)$ consists of a finite set of vertices V, a set of edges E, and a weight function w which maps E into the set of non-negative real numbers.
- A *tree* $T=(V_T \subseteq V, E_T \subseteq E)$ in G is a connected subgraph of G such that the removal of any edge in the subgraph will leave it disconnected. For any $V' \subseteq V$ we say that T *spans* V' in G if $V' \subseteq V_T$
- A tree which spans V in G is called a *spanning tree* of G.
- The weight of a tree T is defined to be $W(T)=\sum \{w(e) \mid e \in E_T\}$
- A leaf of a tree T is a vertex $v \in V_T$ which is connected to one and only one edge $e \in E_T$.

The Steiner tree problem can now formally be defined as follows [28].

Let G denote an undirected weighted graph and let $D \subseteq V$ denote a subset of the vertices in G with $|D| > 1$. A *Steiner tree* T_D with respect to D in G is a tree in G that spans D and has leaves from only the set D.

A *minimum weight Steiner tree* with respect to D in G is defined to be any Steiner tree with respect to D in G whose weight is minimum among all such Steiner trees.

An example showing a weighted graph and the minimum weight Steiner tree spanning the distinguished vertices of that graph is given in figure 8-7.

8.3.2.2. A Steiner Tree Algorithm

The problem of finding a minimum weight Steiner tree given a fixed graph as input is known as the *Graph Steiner Tree Problem.* Since this problem has been shown to be NP-hard [24], heuristic methods have been developed for finding near optimal solutions.

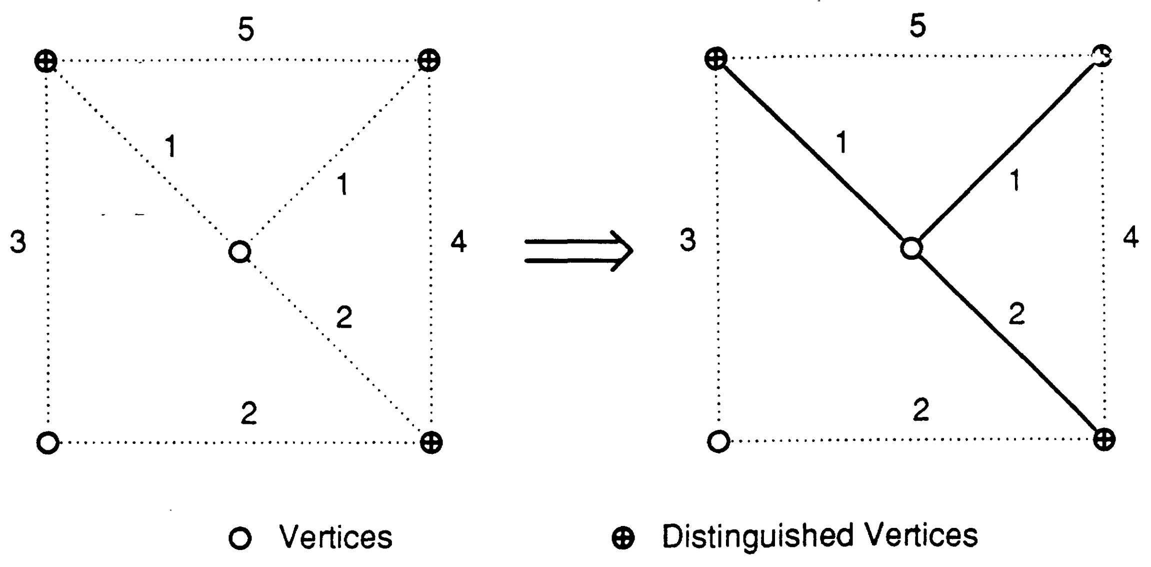

Figure 8-7: A Minimum Weight Steiner Tree

The algorithm used in SOLO and presented here is similar to one developed for use in the global routing phase of the PI placement and routing system [28] [18].

The algorithm is a generalization of Dikstra's single-source shortest path algorithm [29]. An explanation of Dikstra's algorithm will thus be presented first.

Given an undirected weighted graph $G=(V,E,w)$ and some vertex $v_o \in V$ Dikstra's algorithm computes a minimum weight path from v_o to any other vertex in V. Dikstra's algorithm proceeds as follows (taken from [29]).

```
1. $S \leftarrow \{v_o\}$;
2. for each $v \in V$ do
       $C(v) \leftarrow \infty$;
3. $C(v_0) \leftarrow 0$;
4. for each $v \in V-\{v_0\}$ do
       $C(v) \leftarrow w(\{v,v_o\})$;
5. while $S \neq V$ do
       a. Choose a vertex $v' \in V-S$ such that $C(v')$ is minimum;
       b. Add $v'$ to $S$;
       c. for each $v \in V-S$ do
              $C(v) \leftarrow Min[C(v), C(v')+w(\{v',v\})]$
```

This single-source shortest path algorithm can easily be generalized to include multiple sources. It can further be modified to terminate when a minimum weight path to a particular vertex (or to some member of a particular set of vertices) is found.

A modified version of Dikstra's algorithm which, given an undirected weighted graph $G=(V,E,w)$, finds a minimum weight path between some member of the set $T \subseteq V$ and some member of the set $U \subseteq V$ ($T \cap U = \varnothing$) is given below.

```
1. $S \leftarrow T$;
2. for each $v \in V$ do
        $C(v) \leftarrow \infty$;
3. for each $v \in S$ do
        $C(v) \leftarrow 0$;
4. for each $v \in V-S$ do
        for each $v' \in S$ do
                $C(v) \leftarrow Min[C(v), w(\{v',v\})]$
5. while $S \cap U = \varnothing$ do
        a. Choose a vertex $v' \in V-S$ such that $C(v')$ is minimum;
        b. Add $v'$ to $S$;
        c. for each $v \in V-S$ do
                $C(v) \leftarrow Min[C(v), C(v')+w(\{v',v\})]$
6. Return $v'$
7. Return the set of edges included in the minimum weight path.
```

This algorithm returns the vertex of U that is being connected to the vertices of T as well as the edges included in the minimum weight connecting path.

An approximate minimum weight Steiner tree with respect to D in G can thus be constructed by simply the repeated application of this modified algorithm.

The complete minimum weight Steiner algorithm is summarized as follows.

```
1. $E \leftarrow \emptyset$
2. Select some vertex $v \in D$.
3. $T \leftarrow \{v\}$
4. $U \leftarrow D-\{v\}$;
5. **while** $U \neq \emptyset$ **do**
        a. Apply the modified Dikstra's algorithm to the sets $T$ and $U$.
        b. Set $v$ equal to the vertex returned by the modified algorithm.
        c. Set $F$ equal to the set of edges returned by the modified algorithm.
        d. $T \leftarrow T \cap \{v\}$;
        e. $U \leftarrow U-\{v\}$;
        f. $E \leftarrow E \cap F$;
6. Return $E$.
```

This Steiner tree algorithm involves $O(|D|)$ Dijkstra computations, each with time complexity $O(|E| \cdot log|V|)$. This algorithm will thus have a worst case time complexity of $O(|D| \cdot |E| \cdot log|V|)$.

8.3.2.3. Improving the Power Routing

After spanning trees for each of the routing graphs have been been generated, a final optimization step is applied to the Vdd and Gnd routing graphs.

Recall that each power routing graph is initially constructed using only a subset of the edges that would be included in the routing graph associated with a comparable signal net. The set of edges included in each power routing graph is such that the final routing of the graph can be accomplished exclusively in metal regardless of the final routing configuration of the other nets.

Once the global routing of the signal nets has been completed however, it is possible to add some of these previously unusable edges to the power routing graphs while at the same time still ensuring that the final routing of these graphs can be accomplished entirely in a single layer of metal.

This optimization of the Vdd and Gnd routing graphs is accomplished as follows:

1. For the Gnd routing graph:

   Maximally high weights are assigned to those edges in the routing graph that are not included in the spanning tree. This ensures that a newly generated spanning tree will only include edges from the original spanning tree or edges that are subsequently added.

   Edges that had been previously excluded are added to the routing graph if they either:

      a. Do not cross the channel and do not cross any wired power edges.

      b. Do cross the channel but do not cross the wired edges of any net.

   Note that in this context two edges are said to cross if straight lines drawn between the two terminals associated with each edge cross.

   These restrictions are sufficient to ensure that the final spanning graph can still be wired entirely in metal.

   A new minimum weight steiner tree is generated.

2. For the Vdd routing graph:

   The procedure outlined for the Gnd routing graph is repeated.

8.4. Channel Routing

At this point, each channel presents an independent routing problem. The length of each channel is dictated by the placement of the leaf modules. Furthermore, a number of *strands* have been defined for each channel. Each strand defines a set of positions along the edge of the channel that must be interconnected.

8.4.1. Terminology

In this section the basic terminology used in describing a channel routing problem is summarized in detail.

- **Channel**

  The channel is a gridded rectangular region where the routing is accomplished. The size of a channel is generally described in terms of the number of columns and rows it includes.

- **Strand**

  A strand is a list of terminals to be connected within the channel. Although each strand must correspond to a single net, a given net may have more than one associated strand within a channel.

  Each strand may include any number of top or bottom terminals, but is restricted to include at most one left and one right terminal. While a strand is generally implemented by a single horizontal segment and an appropriate number of vertical segments to provide connections to the top and bottom terminals, a strand is occasionally broken into two or more horizontal segments occupying different rows through the use of *doglegging.*

- **Terminal**

  A terminal defines one connection point of a strand with the side of a channel. Each terminal is associated with one side of the channel (left, right, top, or bottom). Those terminals along the top or bottom of the channel each have an associated position which specifies at exactly which column position the connection must be made. Left and right terminals have no associated positions and thus connections to the left and right sides of a channel may be made at any position along the appropriate side.

- **Horizontal Segment**

  A horizontal segment is a segment of interconnecting path that lays on a row of the channel area.

- **Vertical Segment**

  A vertical segment is a segment of interconnecting path that lays on a column of the channel area.

8.4.2. The Channel Routing Problem

A channel router is designed to route the nets that interconnect the terminals on the sides of the channel. The input to a channel routing program consists of a list of strands to be routed as well as a specification of the fixed length of the channel.

A strand consists a set of terminals to be interconnected. When routed, each strand is realized by a connected set of vertical and horizontal segments. It is assumed that only two layers are available for interconnection and that the horizontal segments are constrained to run in one layer while vertical segments must run in the other.

The goal of a channel router is to complete the required interconnections (as defined by a set of strands) using the available number of columns and a minimum number of rows while at the same time minimizing interconnection lengths.

8.4.3. Generating Channel Routing Specifications

In moving between the global routing and channel routing phases of the program, the results of the global router are used to generate an input specification to the channel router for each of the channels in the layout.

Recall that the input to a channel routing program consists of a list of strands to be routed as well as a specification of the number of routing columns available in the channel.

Since the number of columns available in each channel is defined by the placement of the leaf modules in the layout, this input specification can be easily ascertained.

It is also fairly straightforward to generate an appropriate list of strands to define the routing that must be accomplished in each channel. Recall that each channel has associated with it a list of the edges (from the various routing graphs) which are positioned within that channel. It is a simple matter to reduce that list to include only those edges that are part of one of the generated spanning graphs.

While these spanning edges do specify the interconnections that must be realized by the channel routing program, they do not correspond on a one to one basis with the strands that must be used to specify the desired routing. Some strands correspond to single edges whereas others correspond to sets of edges which form connected graphs.

As a result, the list of spanning edges within the channel is next partitioned into a number of sets of edges with each set including those spanning edges which together form a connected graph. The connectivity represented by each of these connected graphs (and thus also by each set of edges) constitutes a strand.

A strand is formed from each corresponding set of edges by simply generating a list including all the terminals (without repeats) associated with all the edges in the set. Applying this process to each of the sets of edges generates the appropriate list of strands for the channel.

8.4.4. Formalizing the Problem

The goal of a channel routing program can be more formally stated as follows. Find an assignment of horizontal and vertical segments such that the interconnection requirements specified by the associated strand set are satisfied and no design rule violations occur.

The design rules which constrain the assignment of horizontal and vertical segments can be summarized as follows.

Two horizontal segments on the same layer, which belong to two different strands and which have in common at least one column, cannot overlap and must be assigned to different rows. This is a *horizontal constraint.*

In the same way, two vertical segments on the same layer, which belong to two different strands and which share the same the same column, cannot overlap. The lowest row occupied by the upper segment must be above the highest row occupied by the lower segment. This is a *vertical constraint.*

The channel routing problem as presented here can utilize an unlimited number of rows. If it is assumed, at least initially, that each strand is implemented with a single horizontal strand, then horizontal constraint violations can always be avoided by simply using a sufficient number of rows. In the worst case a single horizontal segment (and thus also a single strand) would be assigned to each row.

Finding an assignment of segments that will not present any vertical constraint violations is more difficult however.

If it is again assumed that there is only one horizontal segment per strand, then it is clear that the horizontal segment of a strand connected to a top terminal at a given column must be placed above the horizontal segment of another strand connected to the bottom terminal of that column. Such vertical constraints between strands can be represented by edges in a directed graph.

8.4.4.1. Vertical Constraint Graph

The vertical constraints that exist between the various strands in a channel are represented using a *vertical constraint graph.* A vertical constraint graph is a directed graph in which each vertex represents a strand. A directed edge from vertex *i* to vertex *j* means that strand *i* must be placed above strand *j* because of a vertical constraint. A sample routing and the associated constraint graph is illustrated in figure 8-8.

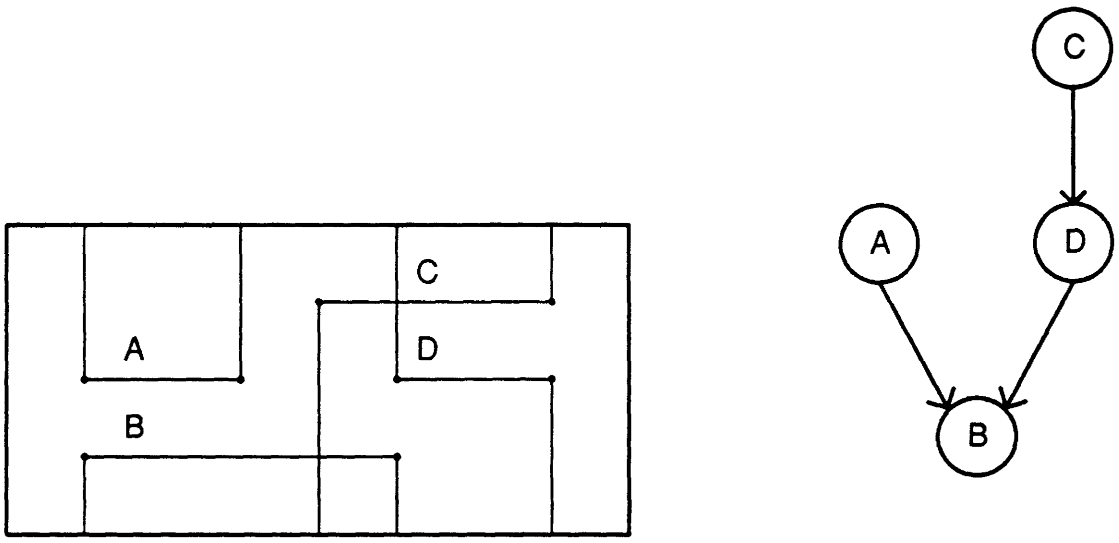

Figure 8-8: A Sample Channel Routing Problem and the Associated Vertical Constraint Graph

If the constraint graph is acyclic, then the routing can be realized by simply maintaining the proper vertical relationships between strands. On the other hand, if there is a cycle in the constraint graph, then a *cyclic vertical constraint* exists and the routing cannot be completed without doglegging.

8.4.4.2. Cyclic Constraints

A routing example which leads to a cyclic vertical constraint is shown in figure 8-9. It is clear that if each strand is constrained to include only a single horizontal segment, then routings such as this cannot be realized.

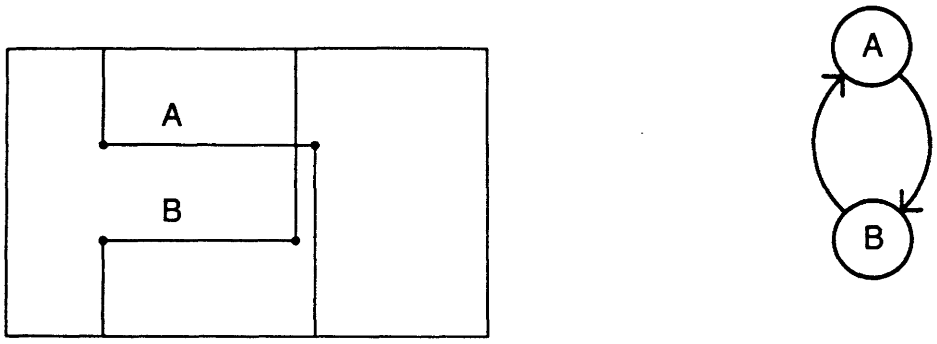

Figure 8-9: An Example with a Cyclic Vertical Constraint Graph

A solution can be found however if strands are allowed to occupy multiple horizontal rows. If this is done, each vertical constraint need only be applied to the horizontal segment connected at the column of interest. The constraint need not be applied to all the horizontal segments that make up a given strand. Allowing multiple horizontal segments per strand can be represented in the constraint graph by splitting each vertex representing a strand into a number of vertices each representing one of the horizontal segments of that strand. This vertex splitting can be used to break cycles and in so doing eliminate vertical constraint violations. This technique is known as *doglegging.*

8.4.4.3. Doglegging

The use of doglegging allows routing channel problems with cyclic vertical constraints to be realized. An example of the use of doglegging in the routing of the cyclic example of figure 8-9 is shown in figure 8-10.

8.4.5. The Algorithm

Although the general channel routing problem has been proven NP-complete [24], various heuristic solutions to the channel routing problem as formulated above have been presented in the literature.

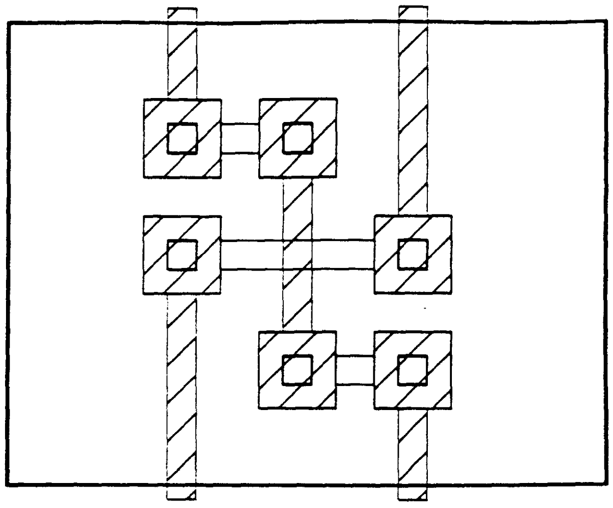

Figure 8-10: Realizing the Routing Through the Use of Doglegging

The *greedy* channel routing algorithm presented by Rivest [30] wires a channel in a left to right column by column manner. Each column is wired completely before the next is started. The algorithm has the advantage that it is quick, simple and it is guaranteed to succeed. The technique does have several disadvantages however. The nature of the technique is such that the interconnection lengths associated with the generated routings tend to be longer than those generated by routers based on more complex graph theoretical algorithms. Perhaps more importantly, the algorithm may under some circumstances have to complete its wiring by the addition of extra routing columns "off the end of the channel".

WEAVER is a knowledge-based channel routing program [31]. The program consists of a number of knowledge-based experts which communicate through a medium called a *blackboard*. Each expert decides which routing step should be taken next and a focus of attention module decides which expert should be allowed to give advice next. In contrast to more standard routing programs, this architecture allows WEAVER to consider a number of different routing metrics in generating its routing solution. Algorithmic approaches to routing are generally forced to consider only one or two of these metrics. The knowledge based approach taken by WEAVER also allows it to eliminate the restriction that horizontal wires must be routed in one layer while vertical wires are routed in another. Thus, WEAVER tends to generate routings of very high quality. It only does so at a very high computational cost however. WEAVER is orders of magnitude slower than algorithmically based routers.

YACR2 (Yet Another Channel Router) is a graph theory based channel router [32]. The program uses one layer for vertical interconnections and another for horizontal interconnections, but allows some exceptions on horizontal interconnections, this feature tending to improve the qualities of its routings substantially. While YACR2 performs as well as or better than any of the other algorithmically based routers on a number of published examples, it too has the disadvantage that additional columns may need to be added "off the end" in order to allow the routing to be completed.

The channel routing method presented here was developed with a number of goals in mind.

- The selected channel routing method must be guaranteed to succeed.
- The computational requirements of the routing program indicate that it should be algorithmically based.
- The routings generated should be comparable in quality (in terms of interconnection lengths) to those generated by graph theory based routers.
- The global routing is done in such a manner as to guarantee that it is *possible* to route the power nets entirely in metal. The channel routing program should ensure that the power routing is indeed accomplished in that way.
- The program should avoid the need to add additional columns "off the end" of the routing channel.
- Finally, the strategy employed by the channel routing program should be based, at least to some extent, on the specific nature of the intended routing environment.

With these constraints under consideration, a graph theory based channel router was developed.

The basic strategy employed by most graph theory based routers involves three phases. First, an initial assignment of segments is generated in which no horizontal constraint violations occur and in which the number of vertical constraint violations is minimized. Next, an attempt is made to remove constraint violations through the use of heuristic doglegging techniques in the remaining available space. Finally, if any constraint violations remain, rows or columns are added and the process is repeated.

This channel router employs a somewhat different strategy.

The first step of the algorithm is to sort the strands into different groups on the basis of the connection requirements of each strand. This step is based on the observation that only strands that are connected to both the top and the bottom of the channel can lead to constraint violations.

Next, those strands with connections to both the top and bottom of the channel are used to generate a vertical constraint graph which summarizes the required vertical relationships among those strands.

If any cycles exist in the constraint graph, an attempt is made to break each cycle by applying a defined set of doglegging techniques to the vertical constraints that makes up that cycle. These defined techniques utilize jogs of at most one column and as such the effects of adding these doglegs to the routing can be represented by the simple addition or removal of edges to or from the constraint graph.

For each cycle that still exists, an additional column is added to provide room for a dogleg to eliminate a vertical constraint and in so doing break that cycle. Each added column is located between two virtual grid lines which define adjacent columns of the channel and at the position in the channel where it is needed. The effects of adding these doglegs can similarly be summarized by the addition or removal of edges to or from the constraint graph.

Finally, the strands are positioned in the channel based on their assigned connection requirement group and as dictated by the now acyclic vertical constraint graph.

The ability to represent the effects of doglegs in the constraint graph allows the selection of the appropriate set of doglegging techniques to be done entirely using the constraint graph representation of the channel. As a result, the final assignment of segments and layout of the channel need only be done once.

There is a disadvantage associated with restricting the repertoire of doglegging techniques to a limited set of single column jogs however. Since many possible doglegging techniques will not be tried, the program may unnecessarily add columns in order to complete a given routing.

The nature of the intended routing environment however lessens the severity of this problem. Each of the leaf modules being placed and routed consists of a linear array of transistors. These transistors are placed such that the gates are positioned on every other grid position with the alternate positions being filled by drains and sources.

Because connections to the drain and source areas of these modules are much less frequent than those to the gates, channel terminals tend to be spaced with free columns situated between them. As a result, single column jogs have a high likelihood of success.

In addition, the cost of adding a new routing column (when necessary) is minimized. Since the routing is accomplished on a virtual grid, columns can be added at any position in the channel. This allows columns to be added where needed rather than always at the end of the channel. No other router is known which allows this flexibility. Furthermore, since the program will never add more than one column between two adjacent grid lines, the distorting effects (on the existing layout) of adding new columns will be minimal.

8.4.5.1. Sorting the Strands

The first step in the routing process is to sort the strands into a number of different groups based on the connection requirements of each strand.

The strands are sorted into the following groups:

- **Top Strands**, which are those strands without any connections to the bottom side of the channel.
- **Bottom Strands**, which are those strands without any connections to the top side of the channel.
- **Through Strands**, which are those strands which connect only to the left and right ends of a channel.
- **Cross Strands**, which are those strands which connect to both the top and the bottom of a channel.

8.4.5.2. Generating the Constraint Graph

Since constraint violations can only occur among the *cross strands*, only these strands are used to generate a vertical constraint graph.

Given a set of strands, the associated vertical constraint graph is generated as follows:

1. A vertex is created to represent each strand.
2. Two linear arrays are created, each with as many elements as there are columns in the channel. One is designated the *top array*, the other is designated the *bottom array*.

3. For each strand:

   - For each top terminal:

     A pointer to the associated strand is placed in the element of the top array that corresponds to the position of the terminal.

   - For each bottom terminal:

     A pointer to the associated strand is placed in the element of the bottom array that corresponds to the position of the terminal.

4. For each element in the top array:

   A directed edge is created between the vertex associated with the strand in that element and the vertex associated with the strand in the corresponding bottom array element. If either element does not contain a pointer to a strand, then no edge is created.

8.4.5.3. Removing Cycles

Before the final assignment of segment positions can be accomplished, any cycles in the vertical constraint graph must be eliminated. Cycles are removed by the use of doglegging techniques to remove edges from the constraint graph.

Only two predefined doglegging techniques may be used for this purpose. One technique involves a single column jog by the strand connecting to the lower terminal at the column of interest. An example of such a dogleg is shown in figure 8-11.

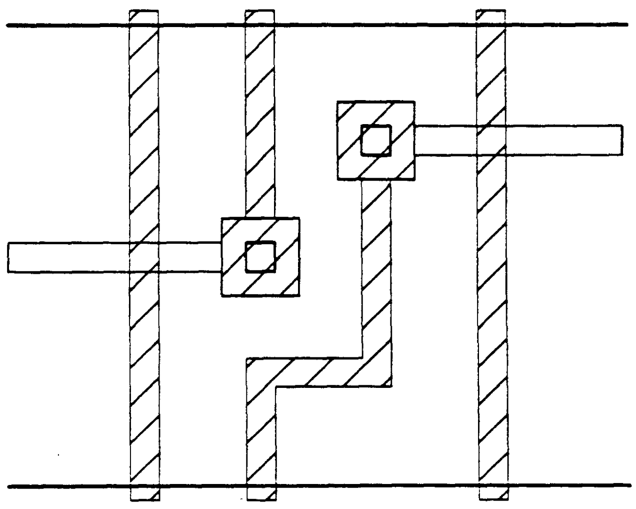

Figure 8-11: A Single Column Jog Dogleg

The other defined technique is similar but it adds a new column to the channel in order to provide room for the dogleg. An example of this type of dogleg is shown in figure 8-12.

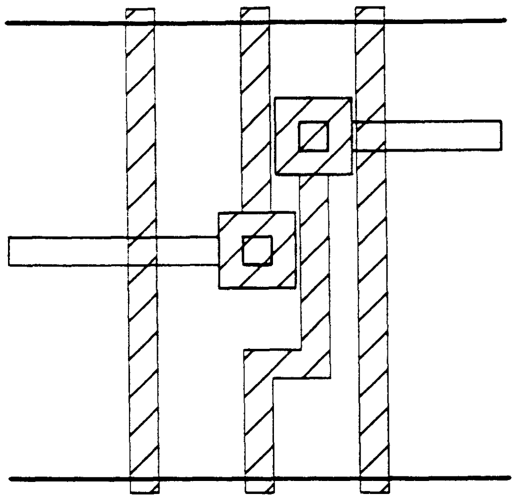

Figure 8-12: A Single Column Jog Dogleg Using an Added Column

It is clear that other techniques could be added to the library of predefined doglegs. Techniques identical to those described above but which instead involve single column jogs by the upper strand are obvious examples. The routing program has performed so well using only these two techniques however that the addition of new dogleg techniques did not seem necessary.

These predefined doglegging techniques provide a means by which edges may be removed from the vertical constraint graph. This ability to remove edges provides a technique whereby cycles in the graph may be eliminated. An obvious question arises however. To which edges should these techniques be applied? Each cycle will contain at least two edges (only one of which need be removed). Furthermore, the removal of an edge from a given cycle and the associated doglegging may affect which other edges (in other cycles) may be removed.

Selecting the proper edges to remove can have a direct effect on the quality of the routing. If doglegging is applied to an inappropriate edge, unnecessary additional rows may be introduced (due to new vertical constraints introduced by the dogleg) or additional columns may be introduced if there is no space available for the desired dogleg. Therefore, an effort is made to to select edges whose corresponding vertical constraints are associated with columns with free space around them.

The first step in the edge selection process is to generate a list of all the combinations of edges which when removed will break all the cycles in the graph. Each of these lists is then scored by summing the scores of the edges when a rough "ease of doglegging" metric is applied. Ease of doglegging is evaluated by looking at at the columns adjacent to the column where the relevant vertical constraint is generated. The score is based on the number of connected terminals in each adjacent column as well as on whether each adjacent column is one to which doglegging is also being applied. Those edges resulting in the highest "ease of doglegging" combined score are selected as those to be removed.

Next, a list summarizing all the possible combinations of doglegging techniques that can be applied to the selected set of edges is generated. Each of these combinations is scored using a procedure similar to that described above for scoring sets of edges to be removed. In this case however, when scoring each edge/dogleg technique pair, only the adjacent column in the direction of the dogleg is scored. A large penalty is associated with those doglegging techniques in which a column is added. More specifically, no combination of techniques in which no columns are added will score below any combination of techniques in which new columns are needed. Furthermore, no combination of techniques which results in the addition of more than one column between two existing columns is allowed. The sets of dogleg combinations are ranked and the highest scoring combination is preliminarily selected.

At this point, the effect of adding the selected doglegs is incorporated into the vertical constraint graph. First, each of the selected edges is removed from the constraint graph. Next, a number of edges and vertices are added to the graph to represent the constraints introduced by the new doglegs.

If there are connected terminals in the adjacent column where the dogleg is positioned, a vertex and three edges are added to ensure that no design rule violations occur. A diagram illustrating the vertex and edges that are added is shown in figure 8-13. The asterisk designates the added vertex. The dummy strand associated with this added vertex prevents a strand with a terminal connection in the adjacent column (where the dogleg must be placed) from being positioned in the row directly below the strand being doglegged around. This ensures that there is sufficient room for the horizontal portion of the dogleg.

If two doglegs share the same column, then still more vertices and edges must be added to the constraint graph. In this case, an added vertex and two edges prevent the

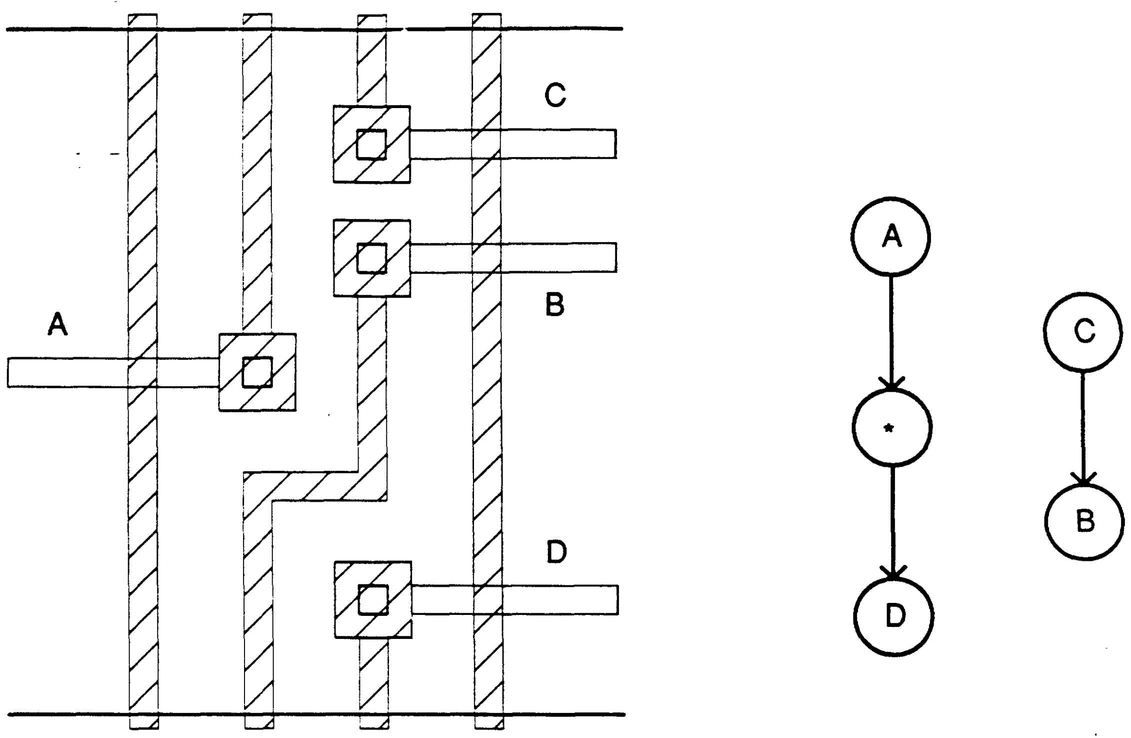

Figure 8-13: Added Edges Associated with a Dogleg and Adjacent Strands

two doglegs from interfering with each other. Figure 8-14 illustrates such a situation along with the added edges and vertex. As in the other diagram, the asterisk designates the added vertex.

It should be noted that these diagrams show only the added edges. It should be clear that other constraint edges will already exist between the illustrated strands. For example, the nature of the doglegs in each diagram makes it clear that an edge from vertex *B* to vertex *A* exists in each case. Furthermore, the two illustrated cases are not exclusive in nature. If two doglegs share a column containing connected terminals, then edges and nodes corresponding to both the situations shown here will be added to the constraint graph.

Since new edges are added to the constraint graph, new cycles may be introduced. Thus, the constraint graph is subsequently checked for cycles. If no cycles exist, a solution has been found and the final assignment stage of the routing begins. If cycles do exist, the constraint graph is restored to its previous state and the next highest scoring combination of doglegging techniques is selected and the appropriate changes to the constraint graph are made. This process is continued until a combination of

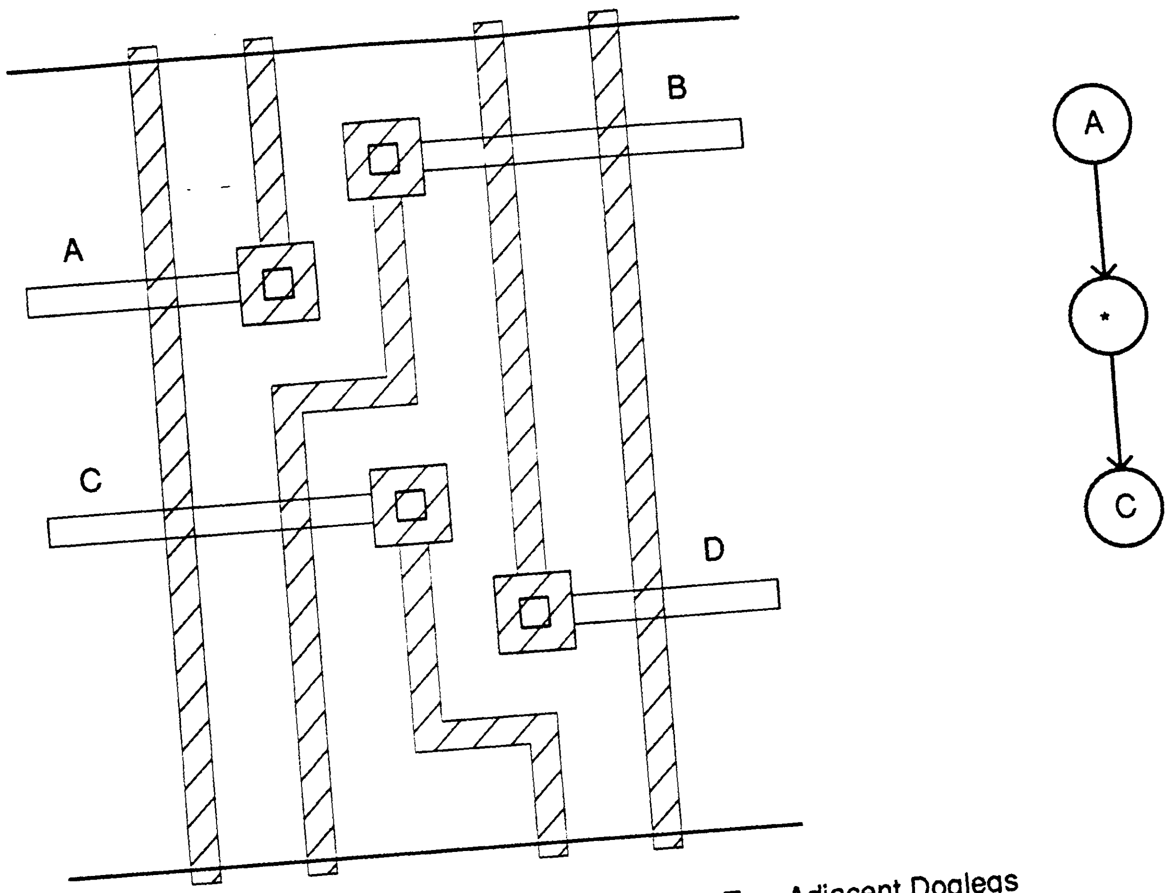

Figure 8-14: Added Edges Associated with Two Adjacent Doglegs

doglegs is found which introduces no new cycles. Since no edges are added for a dogleg in which a new column is introduced, it is clear that this process will eventually terminate.

The procedure outlined here for selecting the set of edges to be removed as well as for selecting the doglegging technique associated with each edge is clearly combinatorial in nature. The routing applications encountered up to this point have all been small enough (in terms of the number of cycles and the number of edges per cycle) however that the computational performance of this part of the routing algorithm has still been quite good. In the future, if the encountered applications become large enough that computational performance becomes an issue, the code will be modified to do a less extensive search of the possible edge and doglegging possibilities when a large number of cycles or edges is encountered. The only effect of this change would be an occasional slight degradation in the quality of the generated routings for large applications.

8.4.5.4. Final Assignment

Once the vertical constraint graph has been made acyclic, the final positions of the strands in the channel can be determined. Strands are positioned in different regions of the channel based on their connection requirement groups. A diagram showing the different areas for the different types of strands is given in figure 8-15.

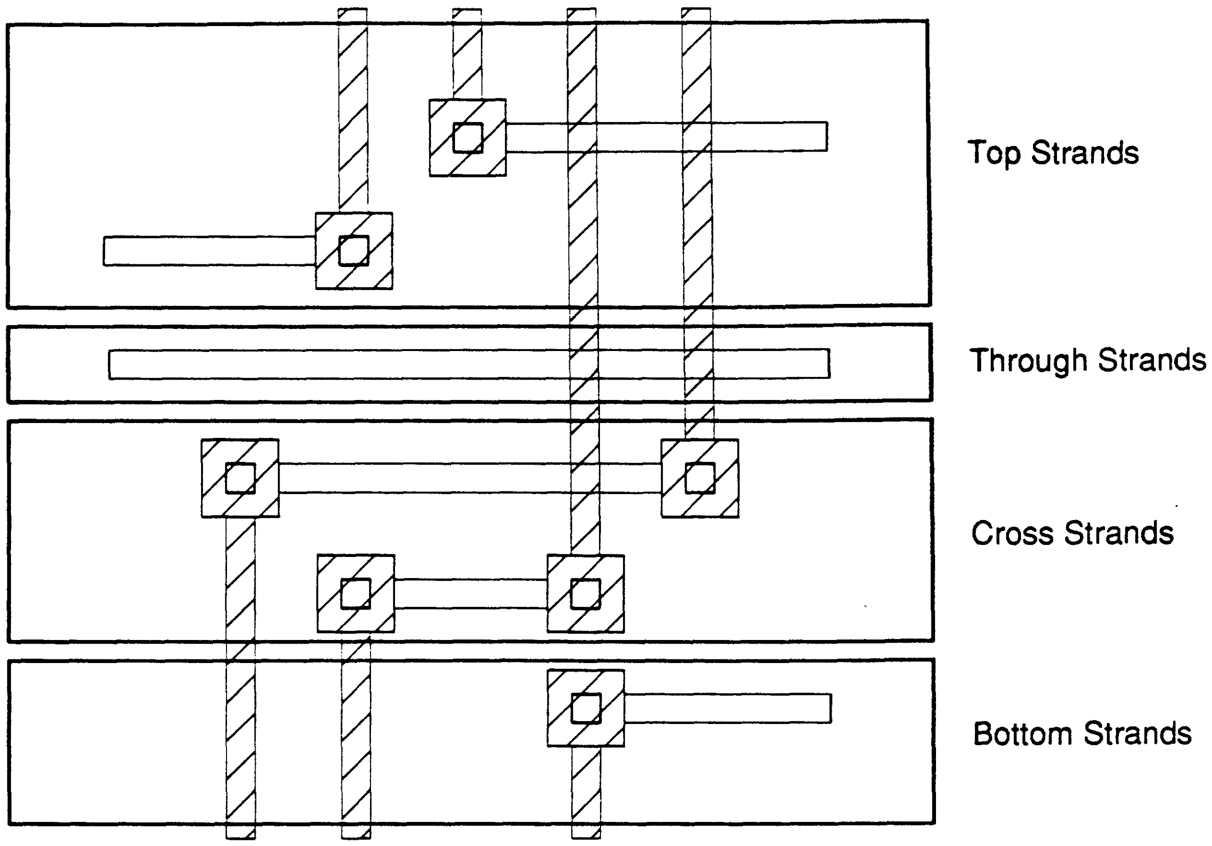

Figure 8-15: The Positioning of Strands within a Channel

The strands are positioned within these regions so as to minimize the number of rows required. In the cross strand region, the strands are positioned such that the positioning constraints dictated by the associated vertical constraint graph are not violated.

Each signal net strand has an associated realization description which specifies the position and extent of its horizontal segment as well as the positions of any connections of that segment to the appropriate top and bottom terminal locations. Each connection is specified to be either a normal straight connection or one of the predefined jogs. The horizontal segment is realized in metal while any connections (including jogs) are realized in polysilicon. The constraints imposed by the vertical constraint graph ensure that the jogged connections will not conflict with any other strands.

Power strands are realized similarly except that both the vertical and horizontal segments of each power strand are realized in metal. Since metal vertical segments cannot cross the horizontal segments associated with other strands (which are also in metal), some special procedures are used to ensure that the metal vertical segments associated with each power strand do not conflict with any other strands.

- In the top and bottom strand regions, the power and ground strands are simply positioned close enough to the associated edge that no conflicts with other strands occur. It is clear that this is always possible to do. Because it is assured that none of the global routing edges associated with the power strands will cross (see section 8.3.2.3), power strands will never conflict with each other. Conflicts with other strands can be avoided by simply positioning the power strands closer to the appropriate edge (either top or bottom) than signal strands.
- Conflicts with cross power strands will not occur. Since the global routing edges associated with cross power strands cross no other edges (also see section 8.3.2.3), conflicts with other strands cannot occur.
- Through power strands have no vertical segments and thus no conflicts can occur.

It should be noted that power strands will never include jogs. Since it is assured that the edges associated with cross power strands will not cross any other wired edges, a cross power strand can never be involved in a vertical constraint.

Although the strands have been positioned so as to minimize the number of rows required for each region, inefficiencies may exist at the interfaces between regions. Thus, after the strands have been placed within their respective regions, a final combining procedure is applied. This procedure merges those rows situated on the edges of adjacent regions which can legally be combined into a single row. This process is illustrated in figure 8-16.

8.5. Complete Routing

This section considers the exact means by which the channel routing algorithm is applied to the globally routed placement.

Recall that one channel is associated with each node of the placement tree. The channels of the placement tree are routed in a depth-first order. As each channel is routed, a composite module is formed consisting of the routed channel and the two children modules of the associated node. Since the exact form of the composite module

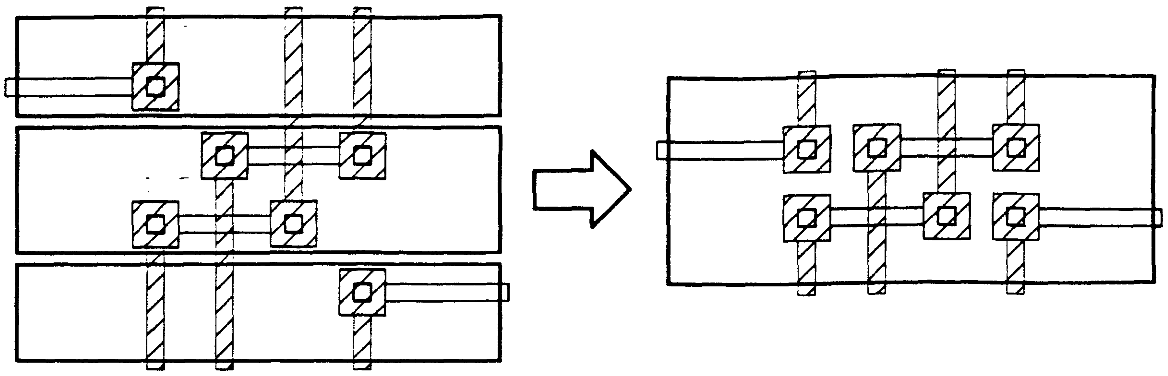

Figure 8-16: Combining Rows on the Edges of Regions

may be slightly different from the form that was predicted by the placement program, the skew parameter associated with each node is readjusted before the final channel routing of the node occurs. This readjustment takes into account any changes in the two children modules.

It should be noted that although the locations of modules (and the associated terminals) may be changed during the channel routing process, the global routing is not recalculated. The selected interconnection paths are thus based on an estimated placement and not on the actual final placement. Since the changes in the placement brought about during the routing process tend to be small, the selected routing paths still tend to be near optimal.

The recursive routing procedure that is applied to the placement tree is outlined below.

1. **If** the left child of the root of the tree is **not** a leaf node
   then channel route the left subtree.
2. **If** the right child of the root of the tree is **not** a leaf node
   then channel route the right subtree.
3. Readjust the skew parameter of the root. The channel routing of the left and right subtrees may have changed the exact locations of terminals on those trees.
4. Channel route the channel associated with the root of the tree.
5. Adjust the spacing parameter of the root to match the number of rows actually used by the routing program.
6. Adjust the locations of the terminals located at the ends of the channel to match the channel end connection locations specified by the routing.

Once all the channels associated with the placement tree have been routed, the four surrounding channels are routed. This completes the routing process.

8.6. Substrate Contacts

In this section, the topic of substrate contact positioning will be considered.

In using a CMOS technology, substrate contacts are needed to tie the substrate and wells to the appropriate supply voltages. At least one such contact is needed for the substrate and one must be included in each well. It is good design practice to distribute substrate contacts much more liberally than this however.

Substrate contact placement is straightforward for automated layout systems which realize CMOS circuits using restrictive layout methodologies such as the gate matrix technique. Since n-MOS and p-MOS transistors are aligned in rows, substrate contacts can simply be positioned at regular intervals along the power line associated with each row. The problem of substrate contact placement becomes much more difficult when a general layout methodology is used.

In the custom layout of CMOS circuits, substrate contacts are typically positioned in the free space between blocks of transistors. It thus seemed that the routing surface between the placed sublayouts would provide the most appropriate place for substrate contacts to be positioned by the program.

A good rule of thumb in CMOS circuit design is to provide one substrate contact for every 5-10 transistors [2]. In terms of the program, providing one substrate contact for each leaf sublayout is thus appropriate. Recall that each sublayout is constrained to only include transistors of a single type.

The substrate contacts situated near n-MOS sublayouts must be connected to Gnd. Conversely, those contacts situated near p-MOS sublayouts must be connected to Vdd. For sublayouts which have connections to Vdd or Gnd (such as pull-up or pull-down networks), the associated substrate contact can be positioned quite easily. It can simply be placed in an adjacent routing channel somewhere along the connecting power line. If a sublayout has no power connections however, the problem of positioning an associated contact can be more difficult.

Since the sublayout has no power connections, it cannot be assured that an appropriate supply line for the substrate contact is situated anywhere nearby. The problem is solved by associating a *dummy* power node with each sublayout that does not include an appropriate power connection. Although no power connections to the sublayout are actually made, the dummy node assures that an appropriate supply line is present in one of the adjacent routing channels.

The complete substrate contact positioning procedure is outlined below. The procedure assures that at least one appropriate substrate contact is situated in close proximity to each leaf sublayout.

1. A dummy node is added to each sublayout which does not include an appropriate power connection. A Vdd node is added to p-MOS sublayouts while a Gnd node is added to n-MOS sublayouts.
2. Four terminals are associated with each dummy node, one positioned at each corner of the corresponding sublayout. These positions are selected since it is assured that there will be no other terminals at these locations and because they allow the dummy node to be accessed on all four sides of the sublayout. Each of these added terminals is given an associated layer of "nil".
3. The standard routing program is applied to the Vdd and Gnd nets with only two minor modifications. Both of these changes relate to how the final channel routing specifications are mapped into layout.

    a. An appropriate substrate contact is added to Vdd and Gnd lines at each position where a horizontal segment of the power routing is connected to a vertical segment which connects the horizontal segment to a leaf sublayout. This will add at least one substrate contact for each sublayout.

    b. If a vertical segment of the routing connects to a terminal with an associated layer of "nil", then that segment is omitted from the layout. This prevents the program from making unnecessary connections to the dummy nodes that have been associated with some sublayouts.

The positioning of substrate contacts is illustrated in figure 8-17. One substrate contact is positioned on a power line connected to a sublayout while the other is positioned near a sublayout with no power connections.

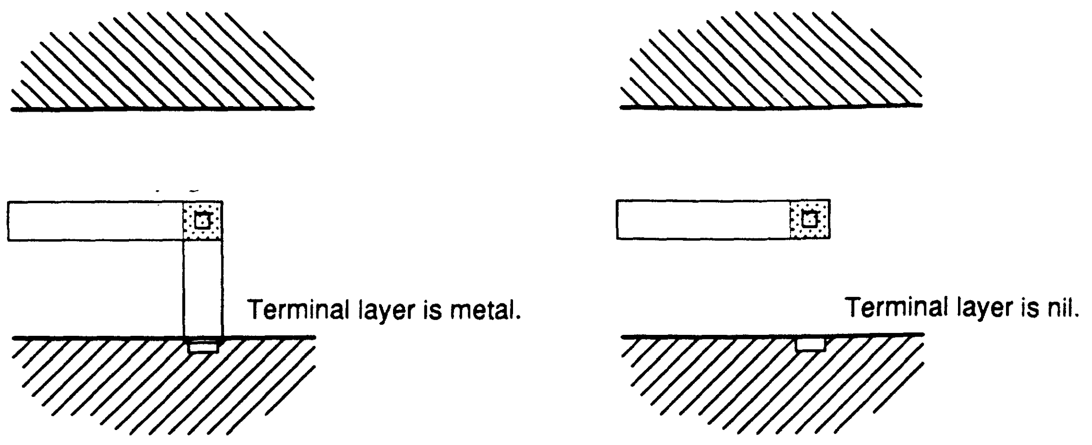

Figure 8-17: Substrate Contact Positioning Near a Connected and an Unconnected Sublayout

Chapter 9
Layout Optimization

Although it is functionally correct, the symbolic layout that exists subsequent to the completion of the routing phase of the program is still somewhat raw in form. This chapter describes the optimization procedures that are applied to the symbolic layout before it is mapped into a mask level representation.

There are two basic aspects of the layout optimization.

One aspect of the optimization involves the removal of unnecessary wires and contacts from the symbolic layout. At this stage of the synthesis process, the layout includes unnecessary objects. Each leaf sublayout is generated with appropriate wires and contacts connecting the transistors of the sublayout to numerous terminal positions situated on its border. Since only some of the available terminals are actually used for interconnection, the wires and contacts connecting transistors to the other terminal locations serve no purpose. These wires and contacts can thus be removed.

The second aspect of the optimization tries to minimize the number of contacts in the layout while at the same time aiming to select interconnection layers that will enhance the circuit's electrical performance. Sometimes these two goals coincide. When a polysilicon wire connected to metal at both ends can be replaced by a metal wire, two contacts are eliminated and the electrical performance of the interconnection is also improved. At other times the goals of contact minimization and layer optimization conflict however. When a metal wire is connected to polysilicon at both ends, it is not clear whether or not the wire should be replaced by a polysilicon one. Although two contacts could be eliminated, the electrical performance of the circuit might be degraded because of the increased resistance of this interconnection path. This optimization phase must thus make decisions relating to the relative desirability of different layout options. This can be difficult to do.

9.1. Removing Unnecessary Wires and Contacts

The first step in the layout optimization is to remove unnecessary wires and contacts. This optimization step involves no subtle decisions as to the relative desirability of different layout characteristics. A simple criterion is applied. Those wires and contacts that are connected to other objects at only a single position on a single layer are unnecessary and can be removed.

The procedure used to remove unnecessary wires and contacts from the symbolic layout is outlined below.

1. The object list defining the symbolic layout is scanned for wires or contacts connected to other objects at only a single position on a single layer.
2. When such an object is encountered, the object is removed and the wires and contacts that had been connected to the removed object are checked to see if they too can now be removed. Those that can be removed are removed and the objects that had been connected to those objects are checked.
3. This *remove and follow* procedure is continued until no more objects in the current search path can be removed.
4. The scan of the object list is then continued.
5. The symbolic layout object list is scanned again and again until a scan is completed in which no object removals are made.

9.2. Contact Minimization and Layer Optimization

The problems of contact minimization and layer optimization have been investigated by many researchers [33] [34]. Although a number of graph-theoretical solutions to the problem have been presented [33], these approaches present problems when applied to the diverse layout environment generated by SOLO. Most significantly, these approaches are unable to deal with more than two layers of interconnect. In addition, these techniques have difficulty basing optimization decisions on any criterion other than simply the minimization of the number of required contacts.

Given these difficulties, a heuristic optimization procedure was developed. The optimization procedure is based on the concept of *clusters* of wires. A cluster is defined to be a connected contiguous set of wires, all on the same layer.

The basic optimization step involves finding clusters that are only connected to other objects through contacts. If all the contacts associated with a given cluster connect to other objects on the same layer and there are no conflicts with other objects, then the wires of that cluster may be replaced with wires of the contact connecting layer and the contacts may be removed. The cluster replacement process is illustrated in figure 9-1.

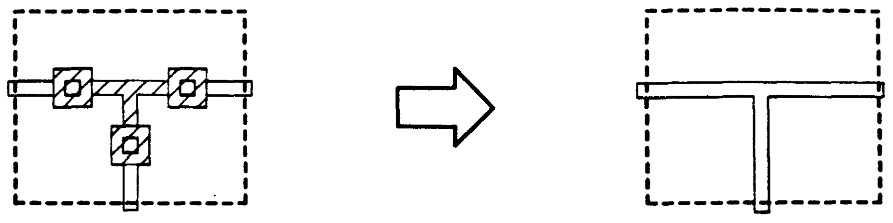

Figure 9-1: Replacing a Cluster and Removing Contacts

Since each step involves both the removal of contacts and a layer change, a decision must be made as to the relative desirability of a particular layer change given that some number of contacts will be removed. The total length[9] of the cluster is used as the decision criterion. The selection of this criterion was based on the idea that if the total length of the cluster is below some maximum value then the cost of the layer change will be more than offset by the associated removal of contacts. In the case of a desirable layer change (such as from polysilicon to metal) the maximum allowable length is set to infinity.

Each phase of the optimization procedure involves scanning the layout for clusters of a particular layer which are suitable to have their wires replaced by those of a particular replacement layer. The wires of those clusters found suitable are replaced and the associated contacts are removed.

[9]The total length of a cluster is defined to be the sum of the lengths of the wires that make up that cluster.

The cluster replacement procedure is described below. This procedure is given the layer of clusters to replace, the intended replacement layer, and the maximum length a cluster may be and still be replaced.

1. The object list defining the symbolic layout is scanned for clusters which are of the appropriate layer and which are connected to other objects only on the intended replacement layer. The cluster must also be such that the replacement of its wires with wires of the intended replacement layer will not conflict with any other objects in the layout.
2. When such a cluster is encountered, the total length of the cluster is calculated.
3. If the cluster is sufficiently small to be replaced, then the wires in the cluster are replaced with wires of the replacement layer and the connecting contacts are removed.
4. The scan of the object list is then continued.
5. The symbolic layout object list is scanned again and again until a scan is completed in which no cluster replacements are made.

The total optimization procedure includes several cluster replacement phases. The exact sequence of replacement phases employed is described below.

1. Appropriate polysilicon clusters are replaced with metal clusters.
2. Appropriate metal clusters are replaced with polysilicon clusters if the total length of the metal cluster is two grid units or less.
3. Appropriate metal clusters are replaced with diffusion clusters if the total length of the metal cluster is six grid units or less.

Chapter 10
A Layout Example

In this chapter an illustrative layout example will be presented. The simple CMOS exclusive-or circuit used throughout the report will be the basis of the example. A schematic of that circuit is shown in figure 10-1.

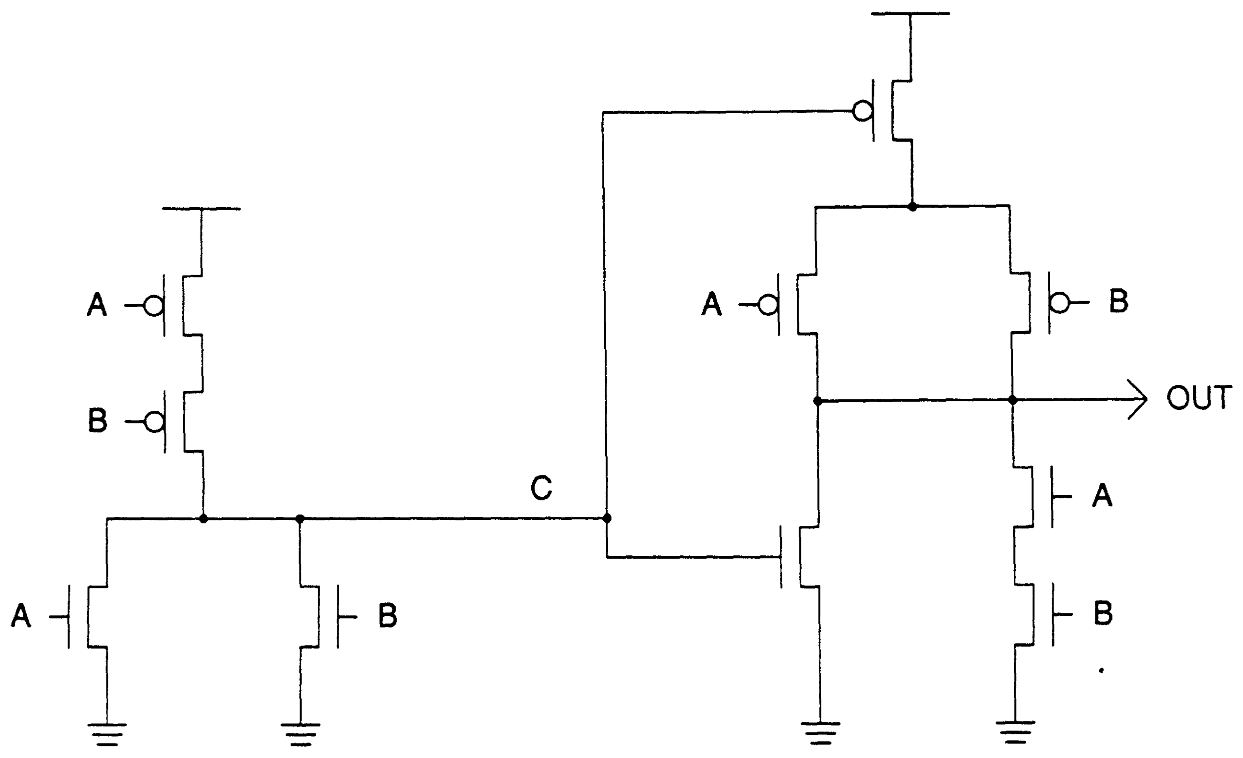

Figure 10-1: Example Exclusive-Or Circuit

10.1. Circuit Specification

The first step in the synthesis process is to create a description of the desired circuit using the circuit specification language. The input to SOLO for this example is shown below.

```
(circuit-macro nor (a b out)
               (series vdd out (a b)
                       :type p :width 24 :length 3)
               (parallel out gnd (a b)
                         :type n :width 8 :length 3))


(defschematic xor (a b out)
  (local c)
  (set-technology cmos21)
  (left a b)
  (right out)
  (nor a b c)
  (series vdd out
          ((transistor (c) :type p :width 24 :length 3)
           (parallel (a b) :type p :width 24 :length 3)))
  (series out gnd (a b) :type n :width 16 :length 3)
  (transistor out gnd (c) :type n :width 8 :length 3))
```

This description specifies the topology of the desired circuit, the desired realization technology, and the size and type of each transistor. The location of the external connections to the circuit are also specified. The inputs *A* and *B* should be located on the left side of the realized circuit while the output *OUT* should be located on the right.

10.2. Restructuring the Hierarchy

The next step in the synthesis process involves a restructuring of the circuit description hierarchy. The original circuit description hierarchy is shown in figure 10-2.

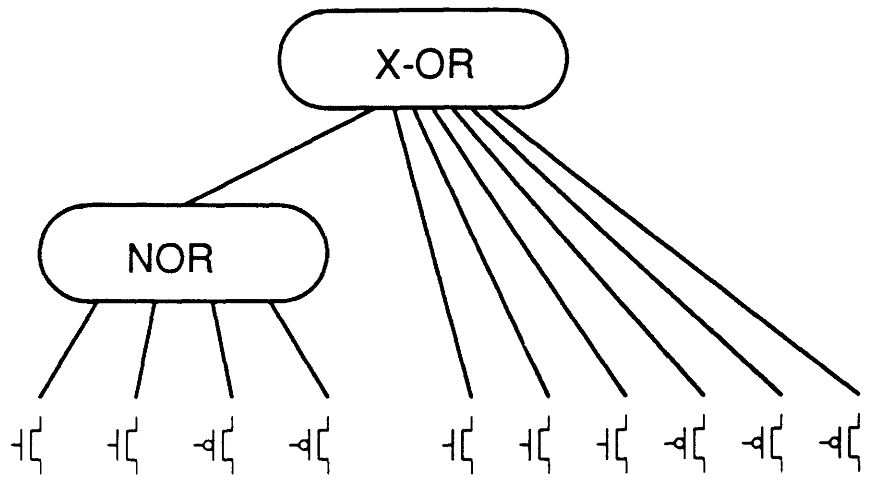

Figure 10-2: Original Circuit Description Hierarchy

The final restructured hierarchy is shown in figure 10-3. This hierarchy illustrates the partitioning of the circuit into series-parallel subcircuits as well as the grouping of subcircuits which constitute pull-up/pull-down pairs.

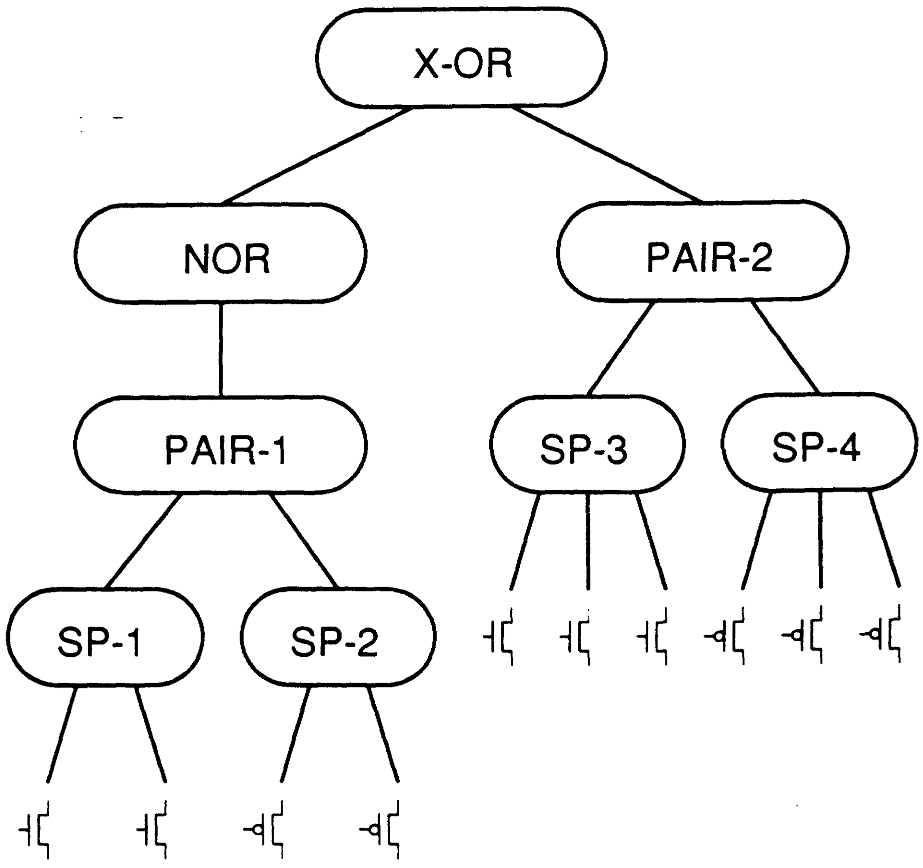

Figure 10-3: Final Circuit Description Hierarchy

10.3. Mapping of Leaves into Layout

After the hierarchy has been restructured, the series-parallel subcircuits which make up the leaves of the new hierarchy are mapped into symbolic layout using a single-row gate matrix methodology. Additional wires are included in these sublayouts which connect externally accessed nodes to the edges of the sublayouts. All connections to the sublayouts can be made in either poly or metal. Poly wires are used to connect to gates while metal wires provide connections to sources and drains. The external connecting wires and internal *personalization* wires for Vdd and Gnd are realized using wider pitch metal wires than those that are used for signal nodes. The exact mappings of the various subcircuits are shown in figure 10-4.

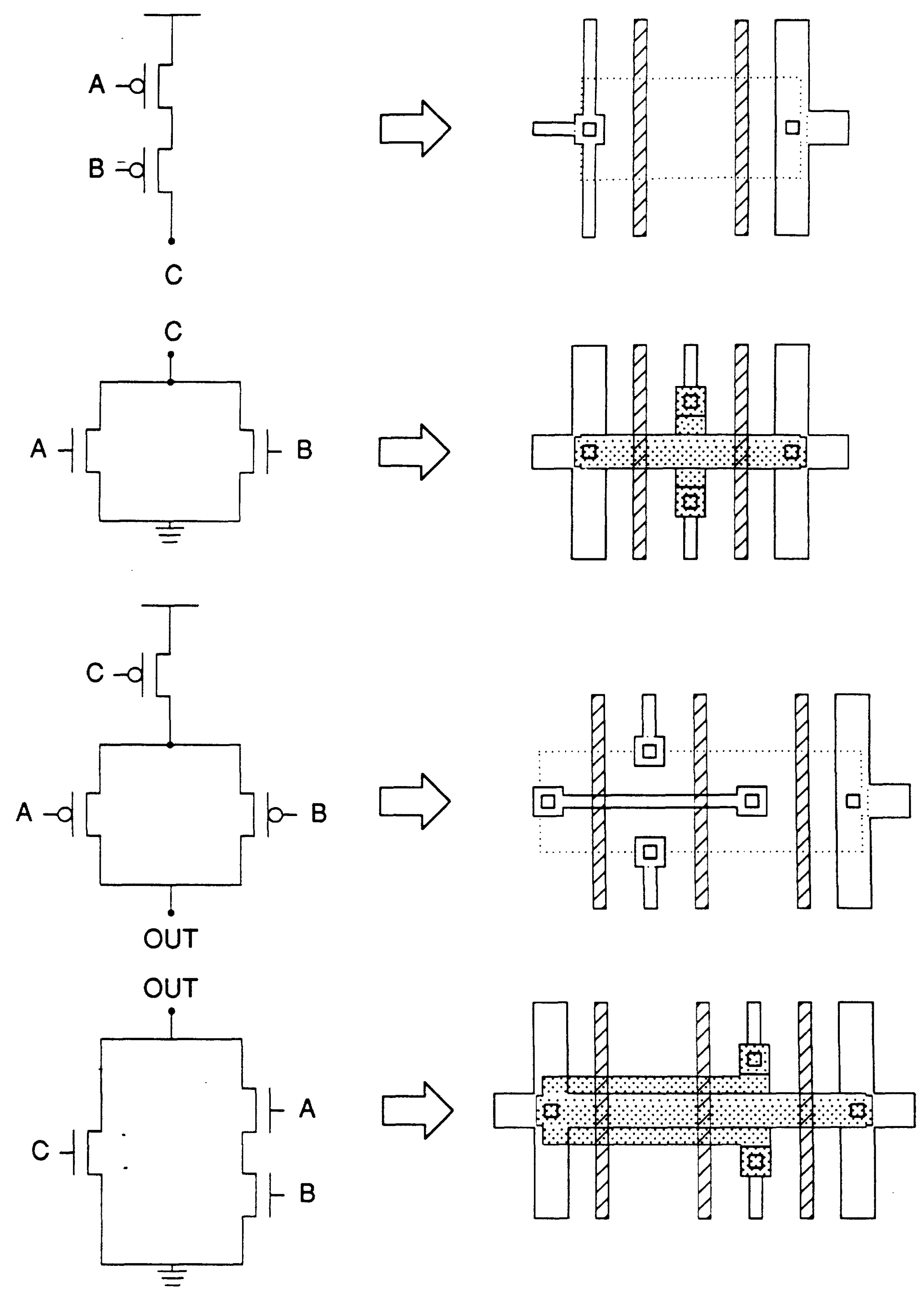

Figure 10-4: Mapping the Leaf Subcircuits into Layout

10.4. Placement

The placement program is next used to place the generated sublayouts in an appropriate manner. During this phase, an exact position, rotation, and mirroring for each sublayout is determined. The positioned sublayouts are shown in figure 10-5. The final placement has clustered the p-MOS sublayouts together and the n-MOS sublayouts together. The p-MOS devices have been situated above the n-MOS devices to provide for efficient power routing given the biased nature of the routing program. Furthermore, the orientations of the placed sublayouts are such that dual n-MOS and p-MOS transistors sharing a common gate are vertically aligned.

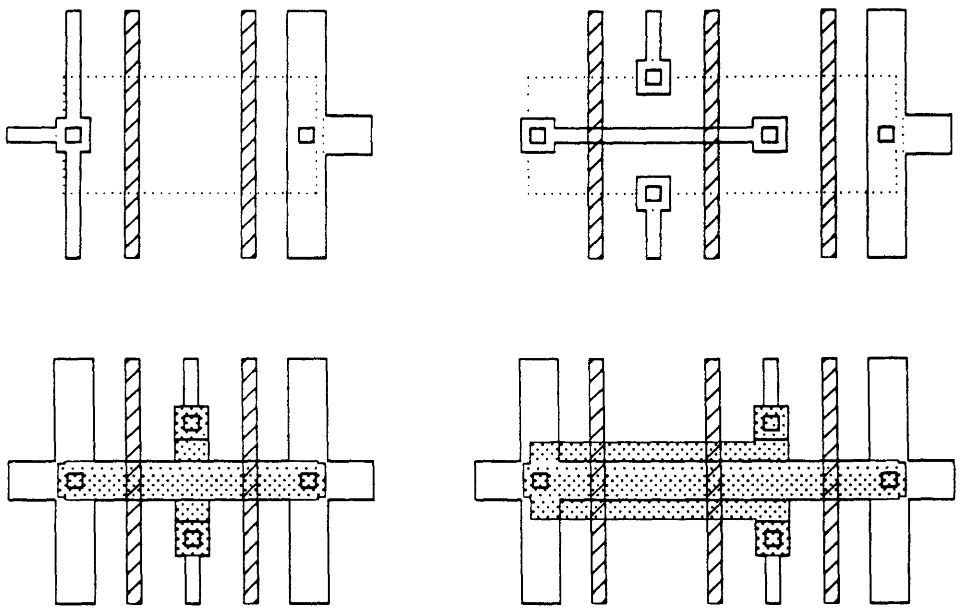

Figure 10-5: The Positioned Sublayouts

10.5. Routing

The actual routing of the placed sublayouts is next performed. The routed layout is shown in figure 10-6.

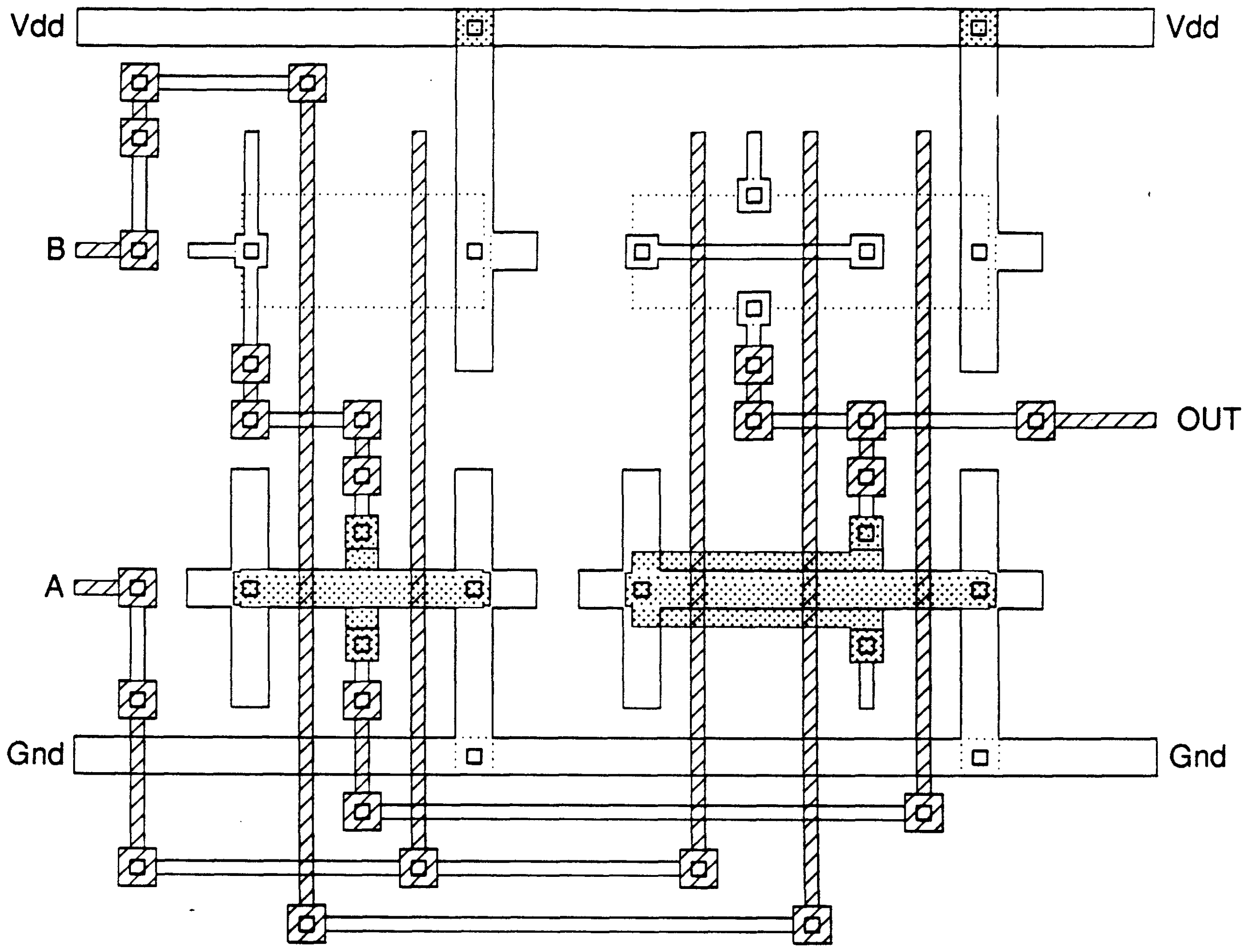

Figure 10-6: The Routed Sublayouts

A Vdd feedthrough is provided across the top of the layout and a Gnd feedthrough is provided along the bottom. This raw symbolic layout illustrates the layout simplifications imposed by the channel routing methodology. Within each channel, horizontal segments are run in metal while vertical segments run in poly. It also shows that only some of the external connecting wires included in each sublayout are actually used. Those that are not needed will be removed during the subsequent optimization phases. Finally, it is interesting to notice that the exact positions of the sublayouts have changed somewhat during the routing process.

10.6. Optimization

Once the raw symbolic layout has been generated, two optimization procedures are applied to improve the quality of the final layout.

The first optimization involves the removal of unnecessary wires and contacts. The symbolic layout which results after this optimization is applied is shown in figure 10-7.

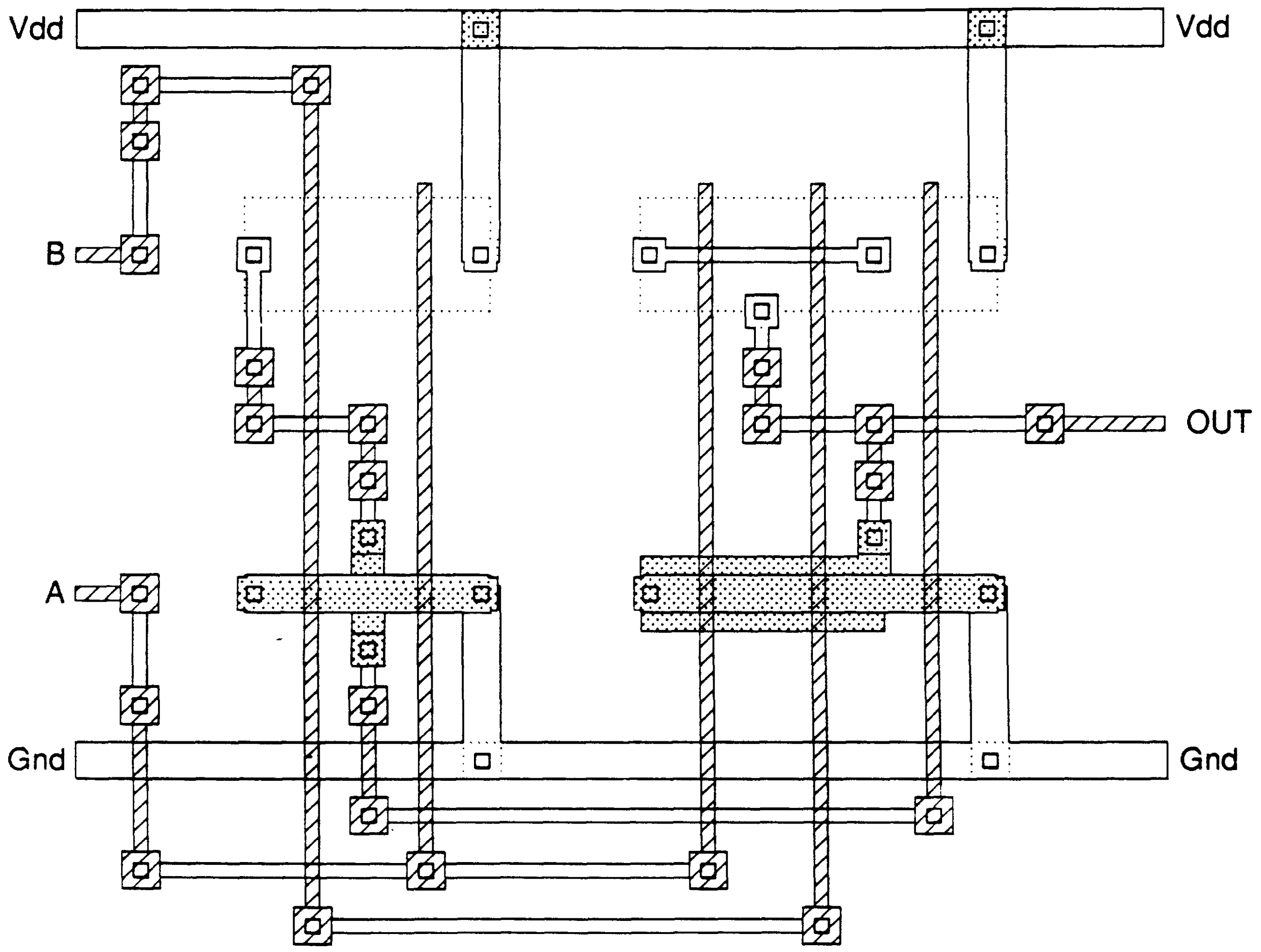

Figure 10-7: After Removal of Unnecessary Wires and Contacts

Next, unnecessary layer changes and contacts are removed. The final symbolic layout is given in figure 10-8.

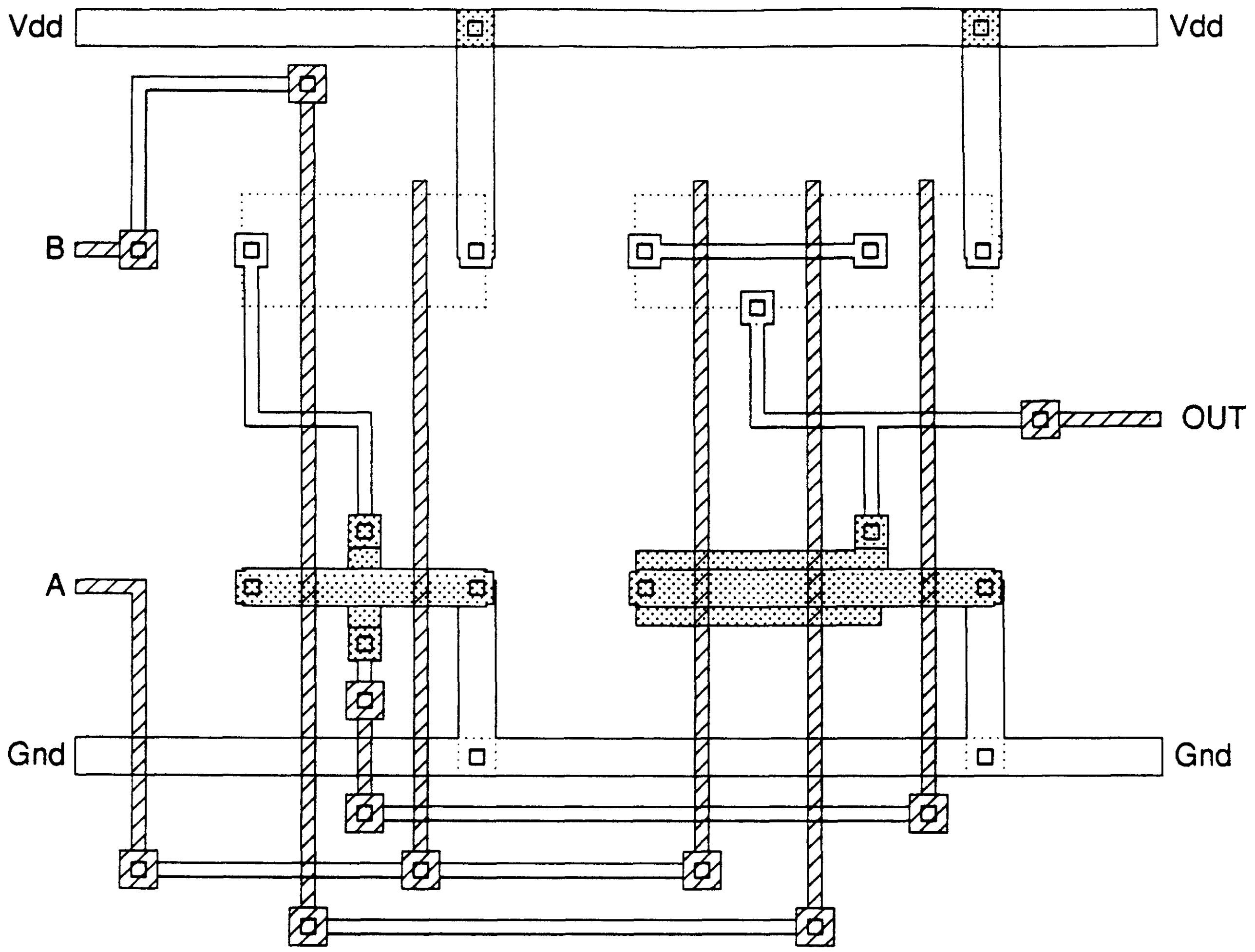

Figure 10-8: Final Symbolic Layout

10.7. Compaction

Finally, a compactor is used to map the symbolic representation into a mask level layout. The final mask level layout is shown in figure 10-9.

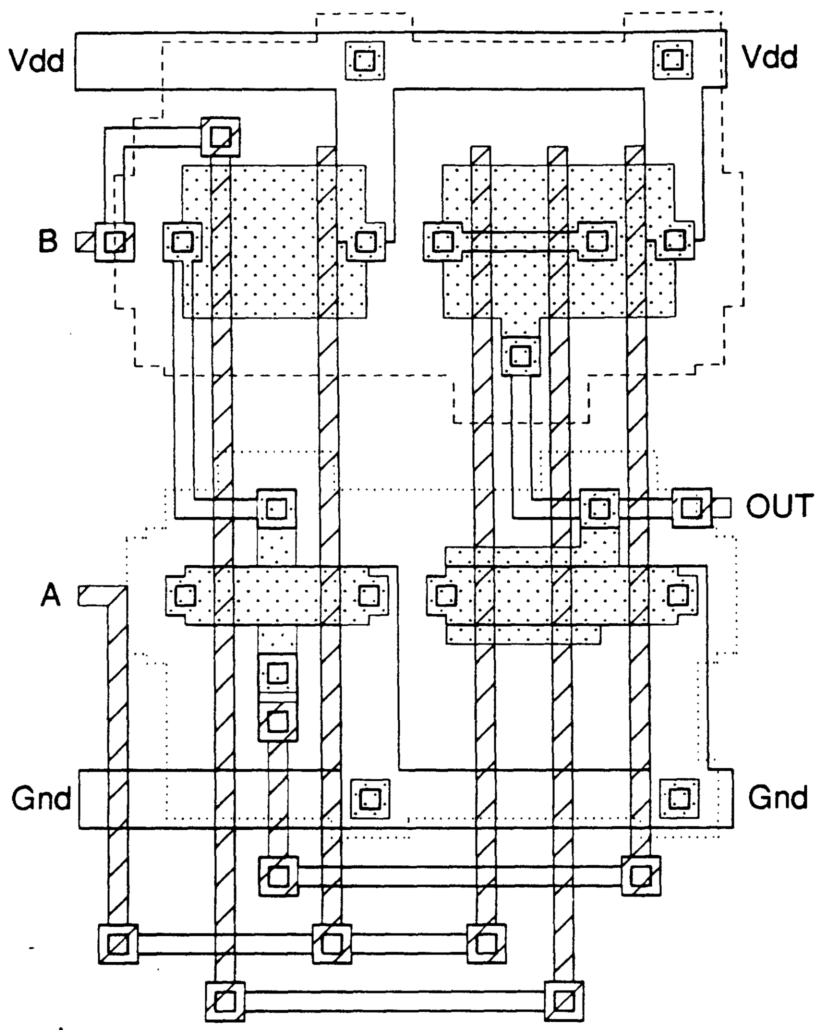

Figure 10-9: Final Mask Level Layout

Chapter 11
Results

In this chapter a number of sample layouts generated by SOLO are presented. Each example includes a circuit schematic and a generated layout. The specification language descriptions corresponding to these examples are given in appendix B.

In all the examples except one, the layouts shown are mask level representations. The mask level layouts were generated using the virtual grid compactor which is part of the NS design system in use at Symbolics Incorporated. [35] The one NMOS example layout is shown in symbolic layout form because a compactor compatible with NMOS designs has yet to be acquired.

Finally, an analysis of these results is presented in terms of the original design goals for the program.

11.1. Layout Examples

11.1.1. Exclusive-Or Circuit

The exclusive-or circuit analyzed extensively in chapter 10 is presented again for completeness. This example demonstrates the ability of the program to effectively utilize a gate matrix layout methodology when such a restrictive methodology is appropriate. A schematic of the circuit is given in figure 11-1. The corresponding layout is shown in figure 11-2.

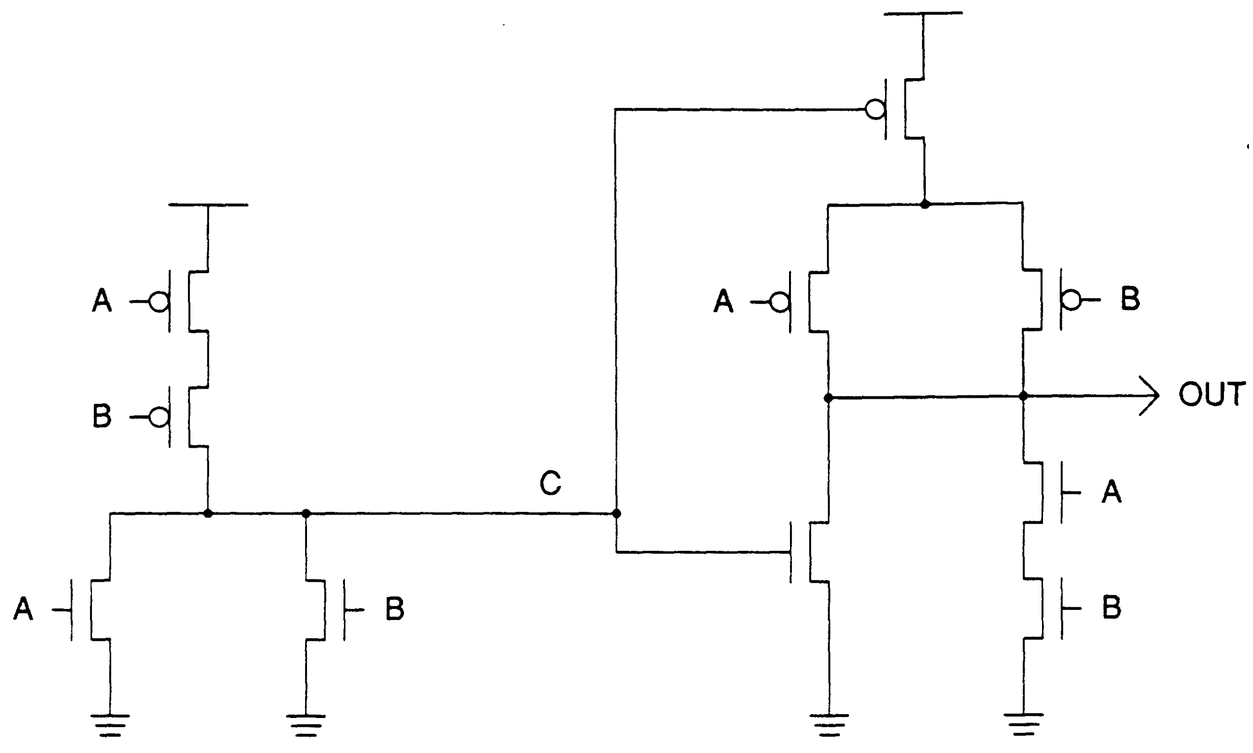

Figure 11-1: Exclusive-Or Circuit

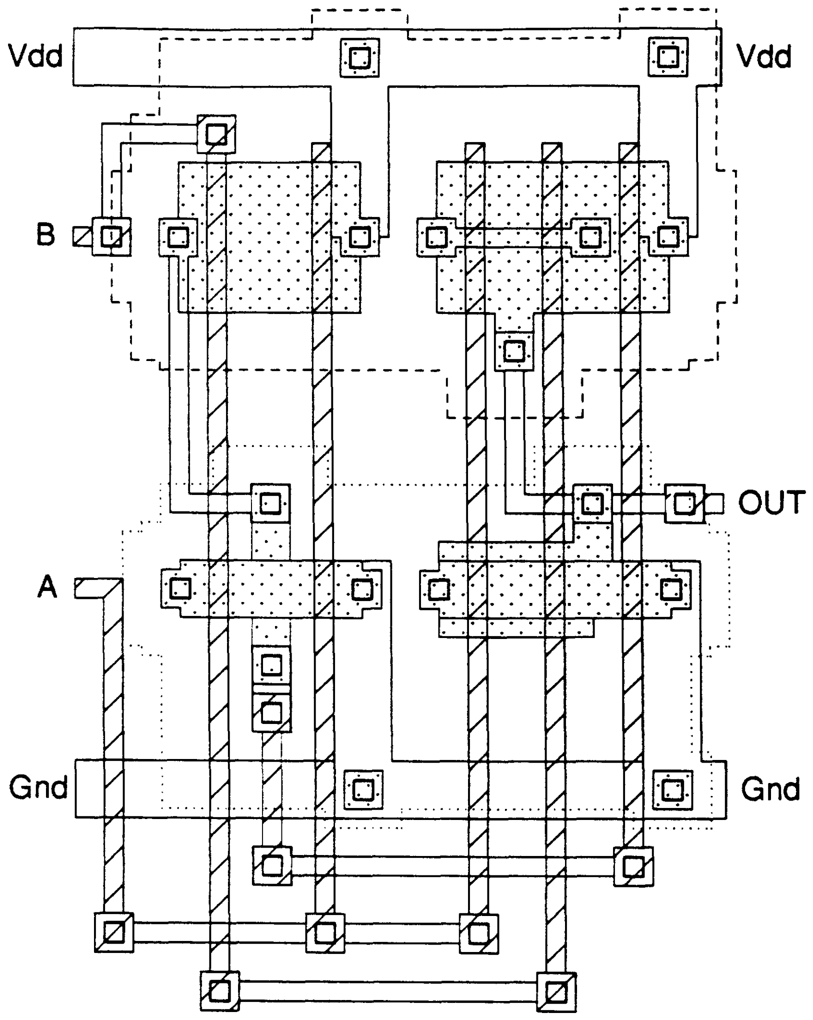

Figure 11-2: Exclusive-Or Layout

11.1.2. Clocked Exclusive-Or Circuit

The circuit presented here is a clocked CMOS realization of an exclusive-or circuit. The example illustrates the ability of SOLO to efficiently implement circuit methodologies which include unequal numbers of n-type and p-type transistors. The example also demonstrates the use of feedthroughs for the effective realization of clocked circuit methodologies. The circuit schematic is given in figure 11-3. The generated layout is given in figure 11-4.

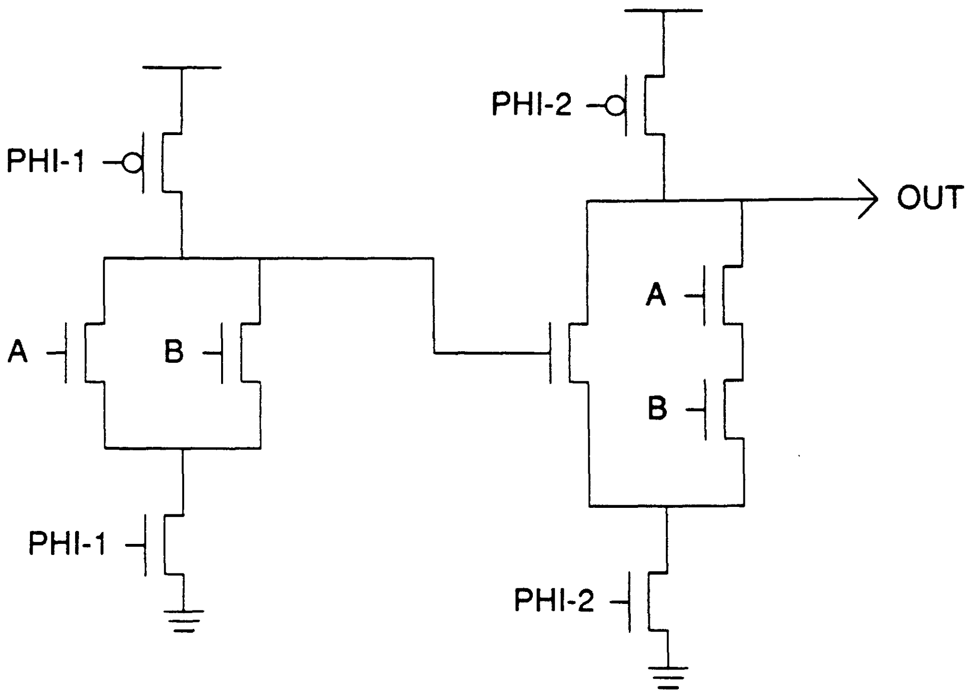

Figure 11-3: Clocked Exclusive-Or Circuit

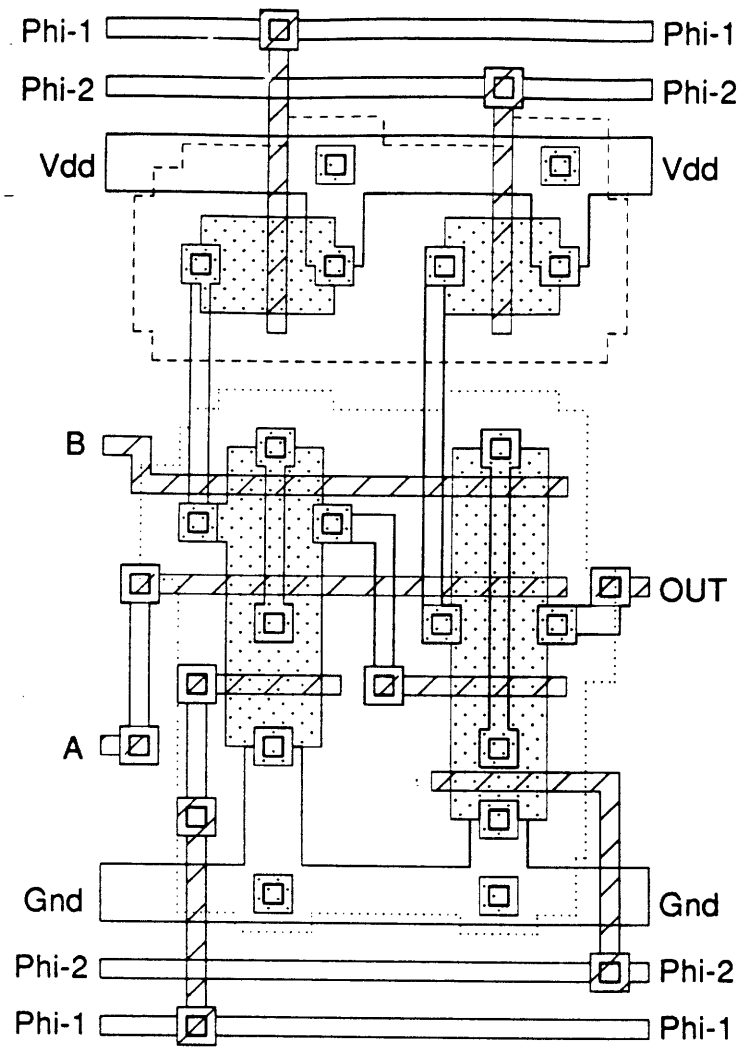

Figure 11-4: Clocked Exclusive-Or Layout

11.1.3. Classical CMOS Adder Circuit

The circuit shown in this example is classical CMOS adder circuit. The schematic is given in figure 11-5. The circuit was realized several times each with a different desired aspect ratio. The layout shown in figure 11-6 was generated with no specified aspect ratio. The layout in figure 11-7 was generated with a specified aspect ratio (width to height) of 1 to 4. The layout given in figure 11-8 was generated with a desired aspect ratio of 4 to 1.

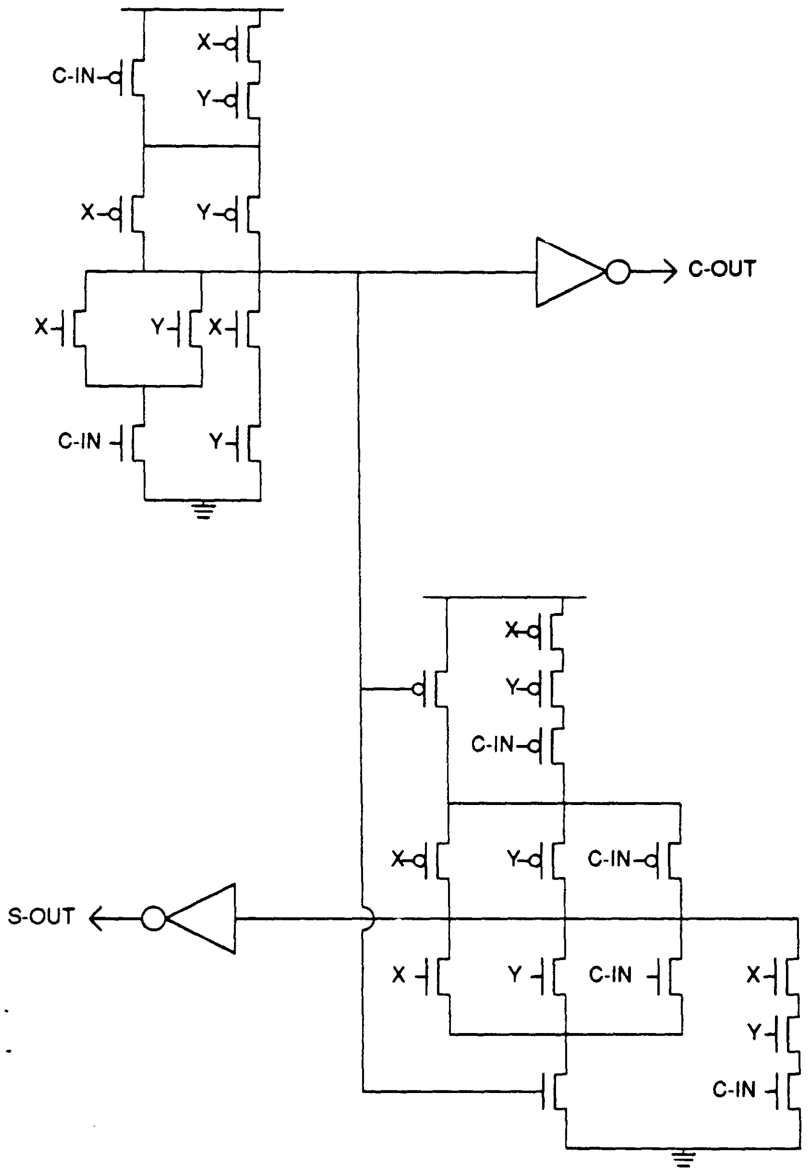

Figure 11-5: Classical CMOS Adder Circuit

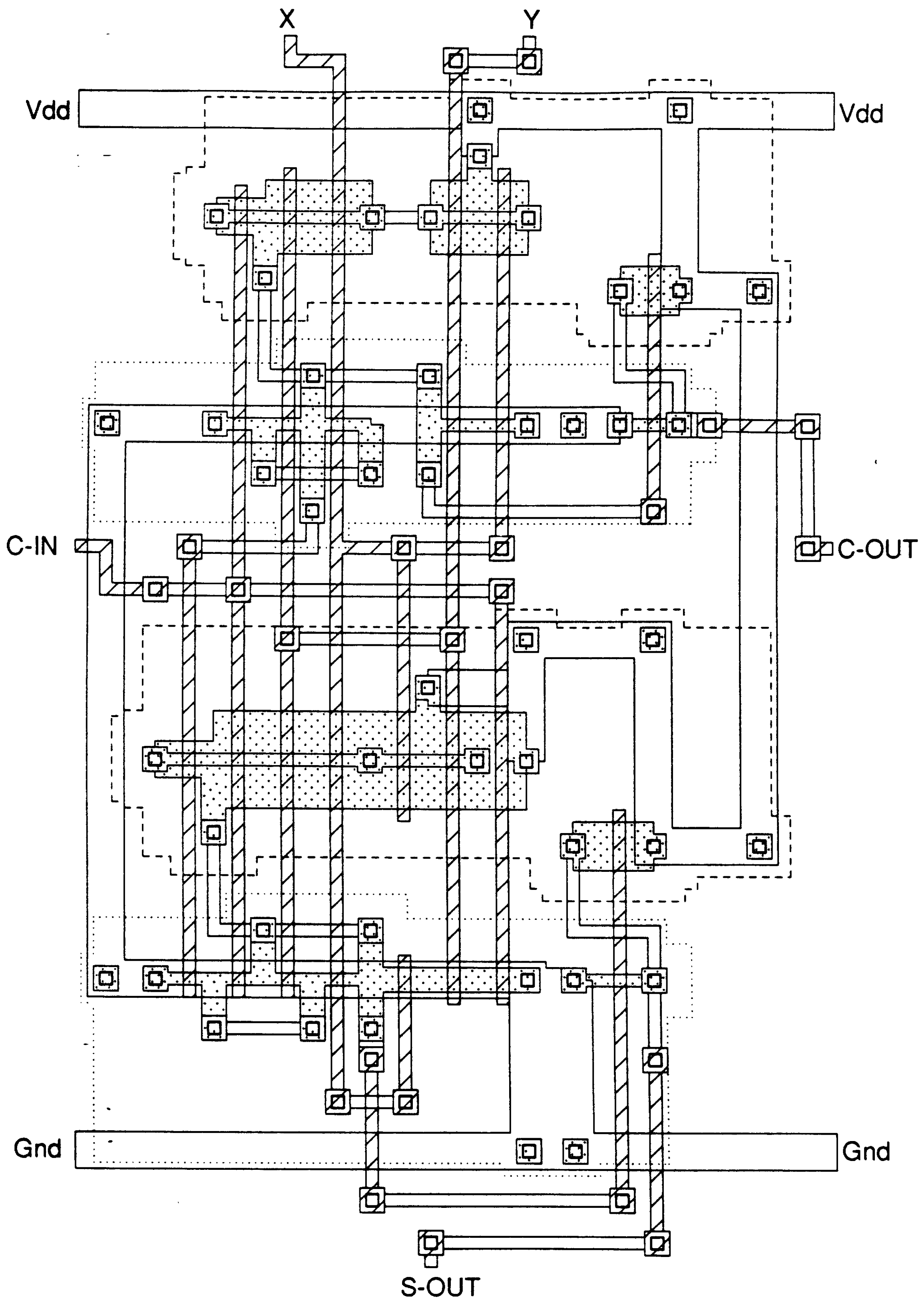

Figure 11-6: Classical CMOS Adder Layout

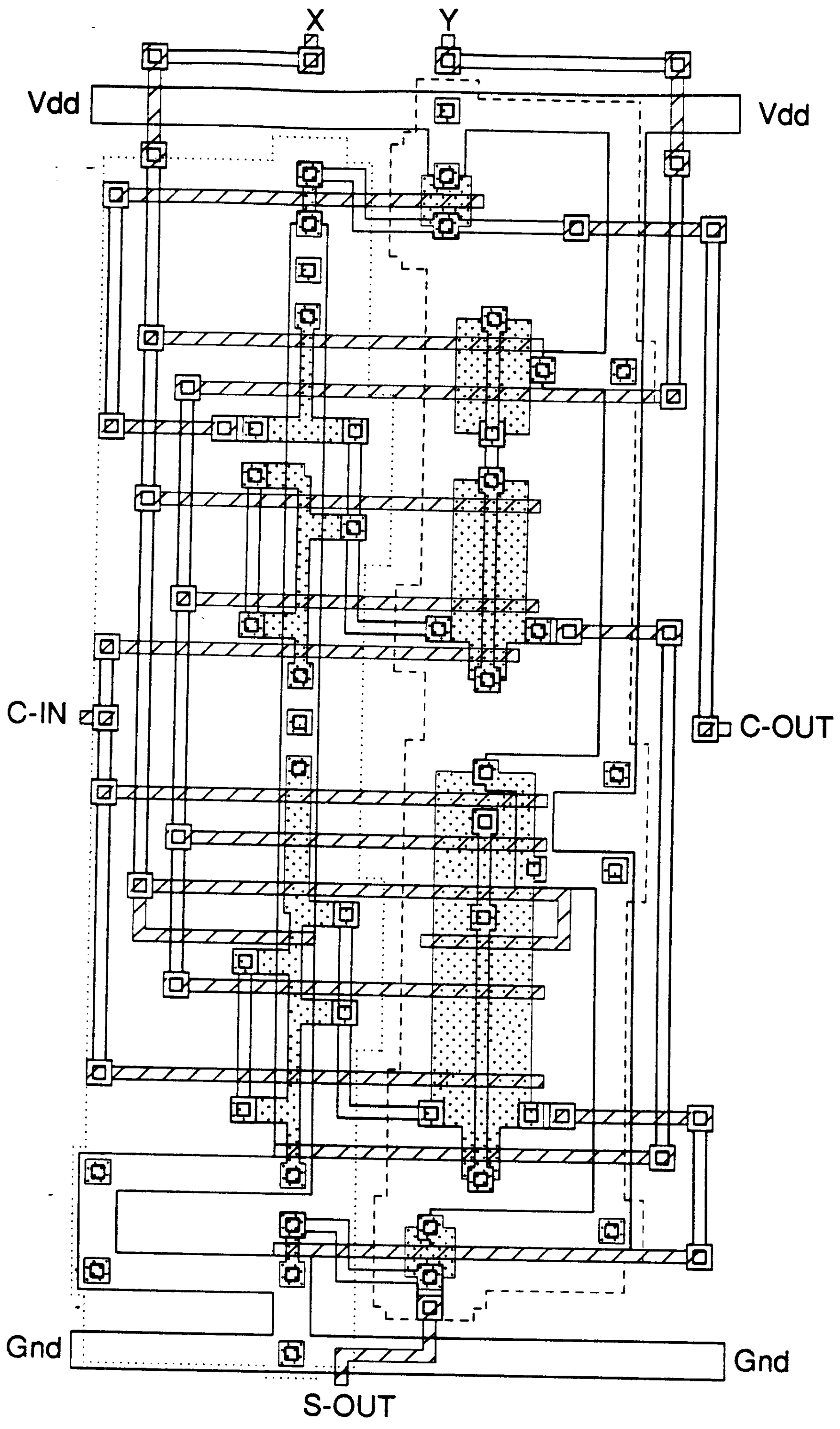

Figure 11-7: Classical CMOS Adder Layout with Desired Aspect Ratio of 1 to 4

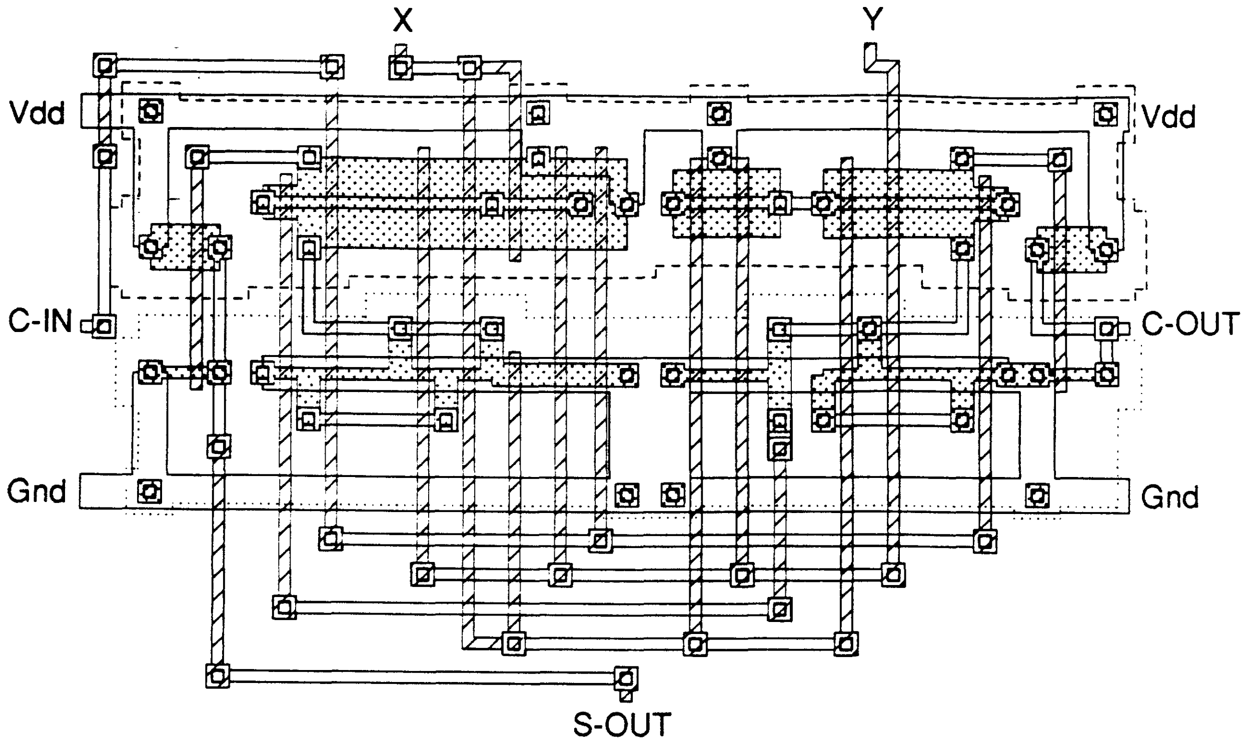

Figure 11-8: Classical CMOS Adder Layout with Desired Aspect Ratio of 4 to 1

11.1.4. Pass Transistor Adder Circuit

This example presents the realization of a well known transmission gate adder circuit. The circuit schematic is given in figure 11-9. The example demonstrates the problems SOLO has when applied to a circuit which does not partition effectively. The series-parallel partitioning methodology does not deal well with pass transistor circuits. The circuit is partitioned into subcircuits which include exactly one transistor each! The generated layout is shown in figure 11-10. Despite the poor partitioning, this example does demonstrate the program's ability to effectively cluster n-MOS and p-MOS sublayouts.

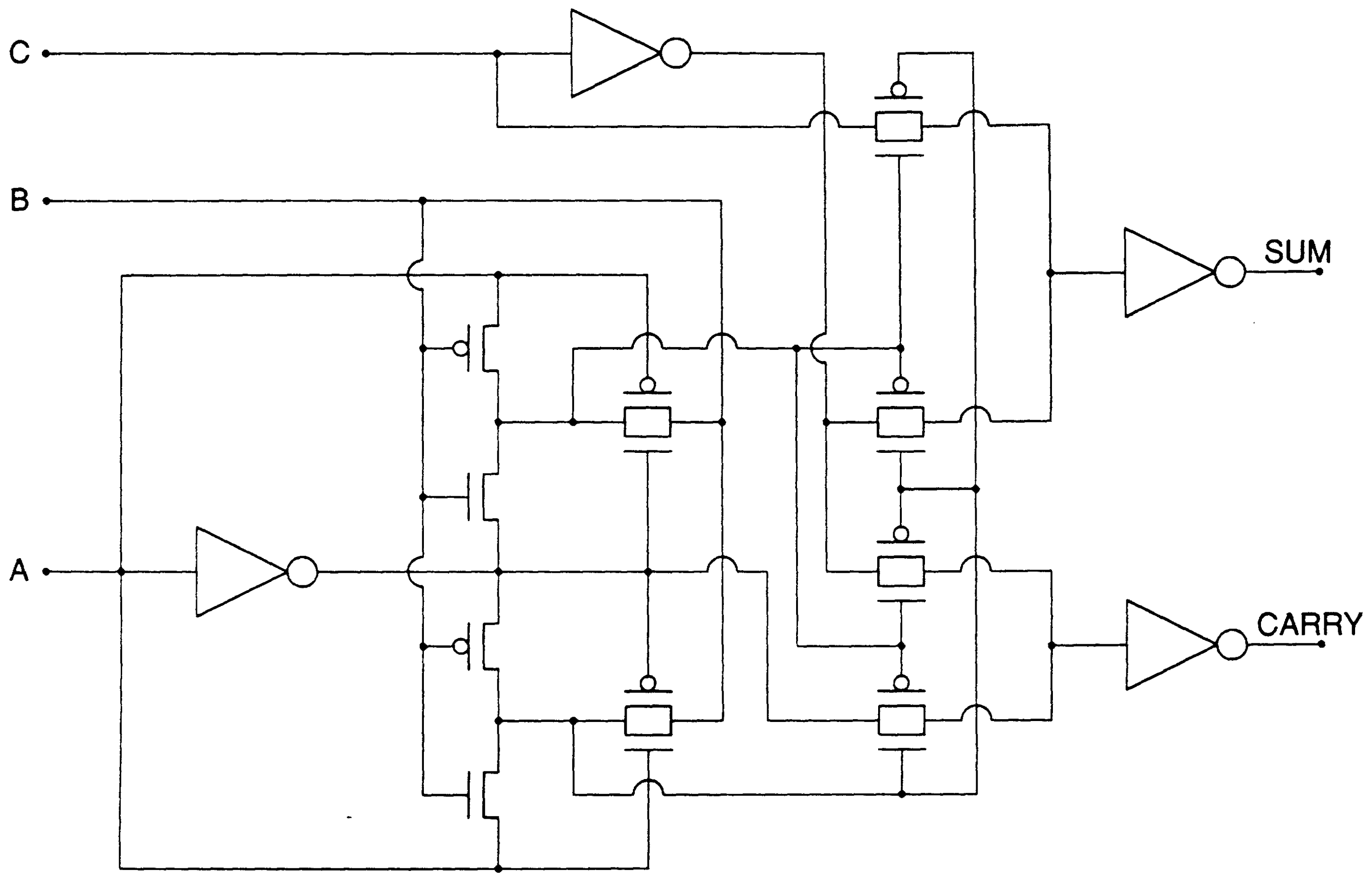

Figure 11-9: Pass Transistor Adder Circuit

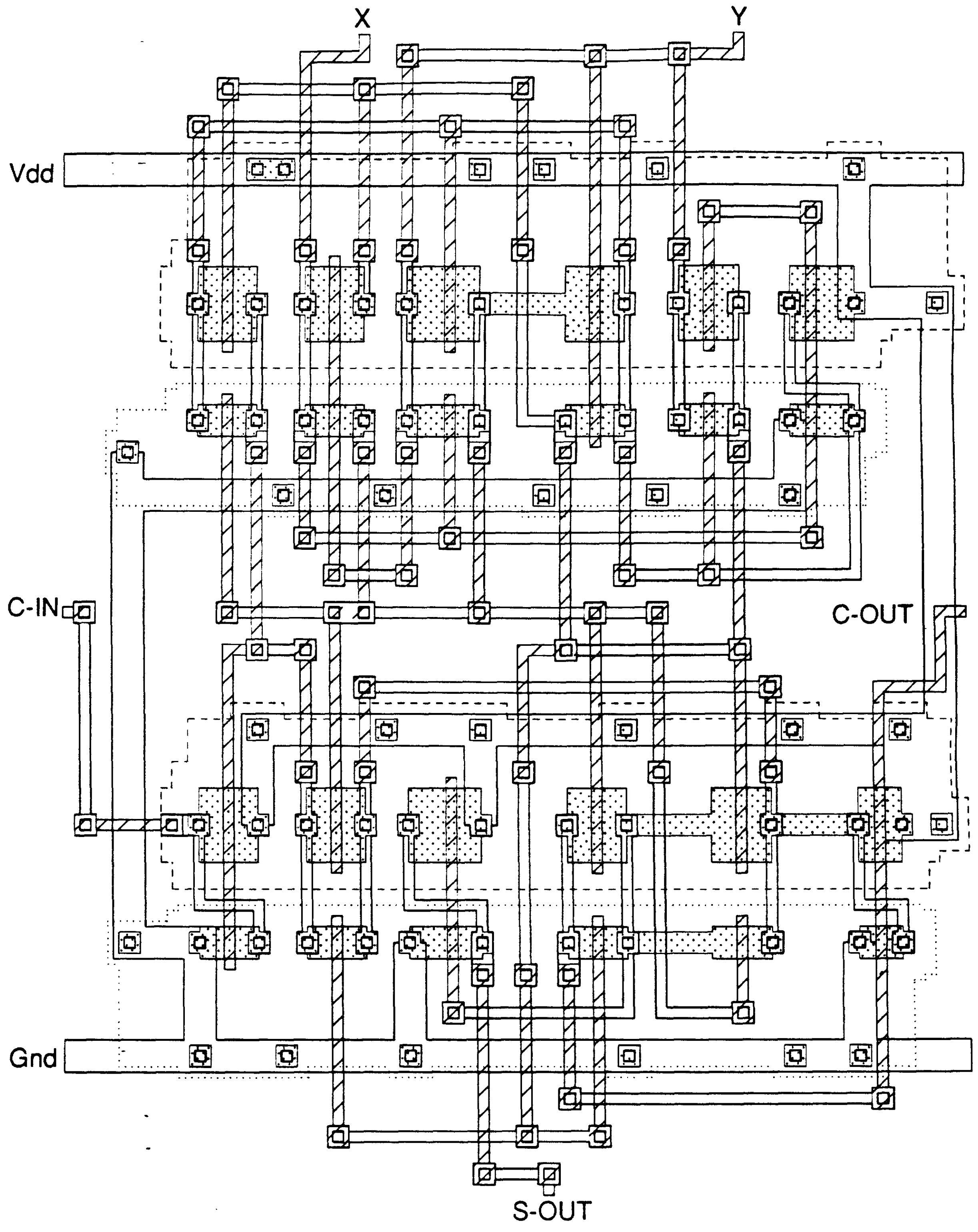

Figure 11-10: Pass Transistor Adder Layout

11.1.5. D Flip-Flop Circuit

SOLO can also deal with the layout issues involved in the realization of very small circuits. The D flip-flop circuit shown in figure 11-11 includes only 6 transistors. The corresponding layout is given in figure 11-12.

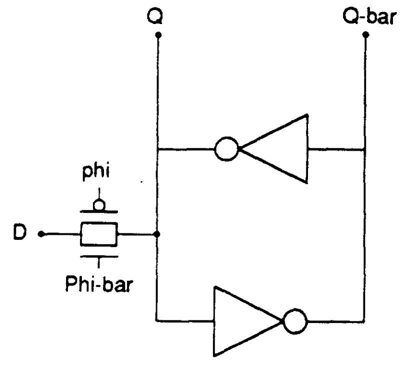

Figure 11-11: D Flip Flop Circuit

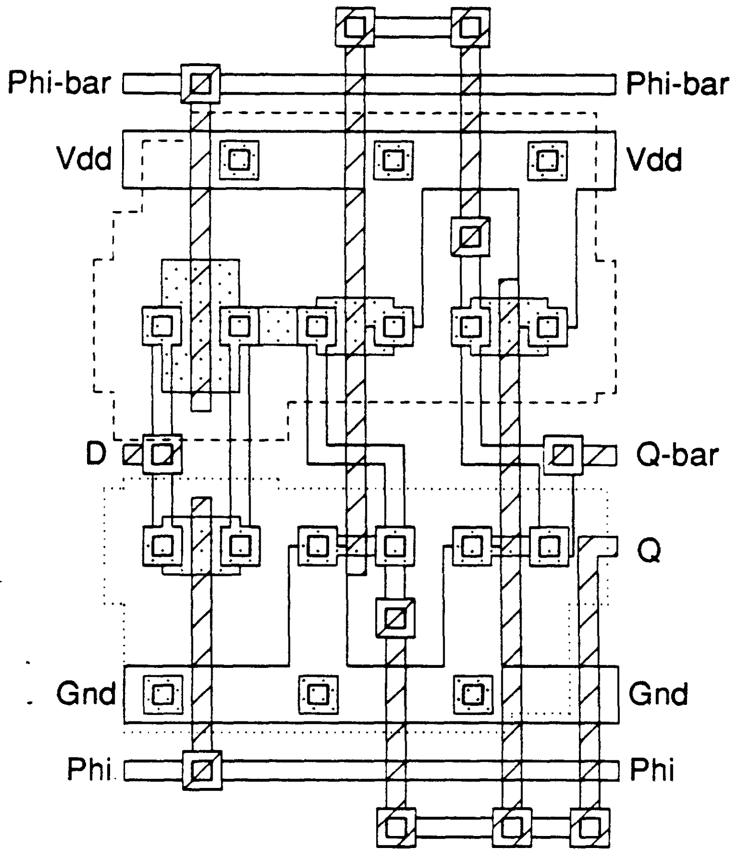

Figure 11-12: D Flip Flop Layout

11.1.6. Four-Input Exclusive-Or Circuit

The circuit presented in this example consists of three instances of the two-input static exclusive-or circuit already presented. These circuits are interconnected so as to form a four-input exclusive-or. The interconnection scheme is shown in figure 11-13. The output exclusive-or is realized with wider transistors in order to accommodate a large output load. This example demonstrates the program's ability to realize circuits including transistors with widely differing gate widths. The generated layout is shown in figure 11-14.

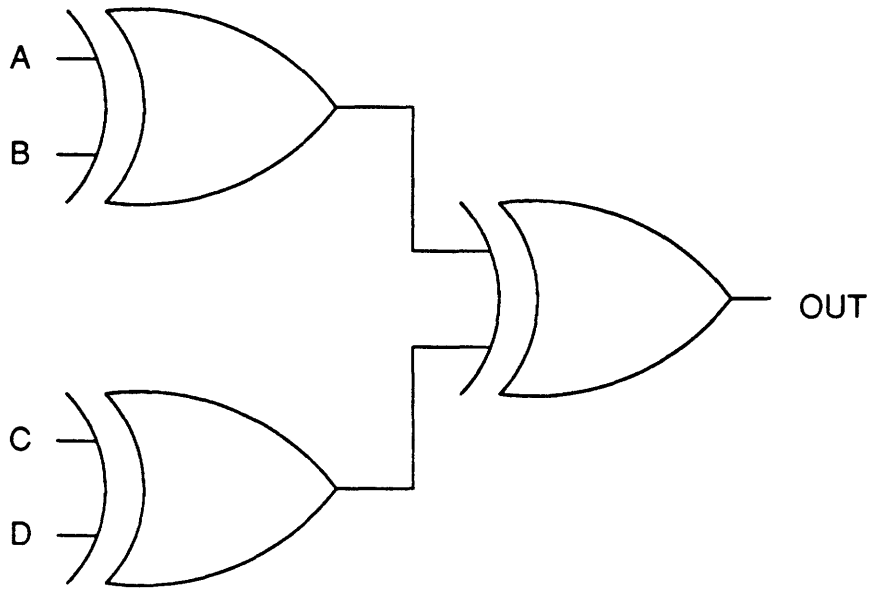

Figure 11-13: Four Input Exclusive-Or Circuit

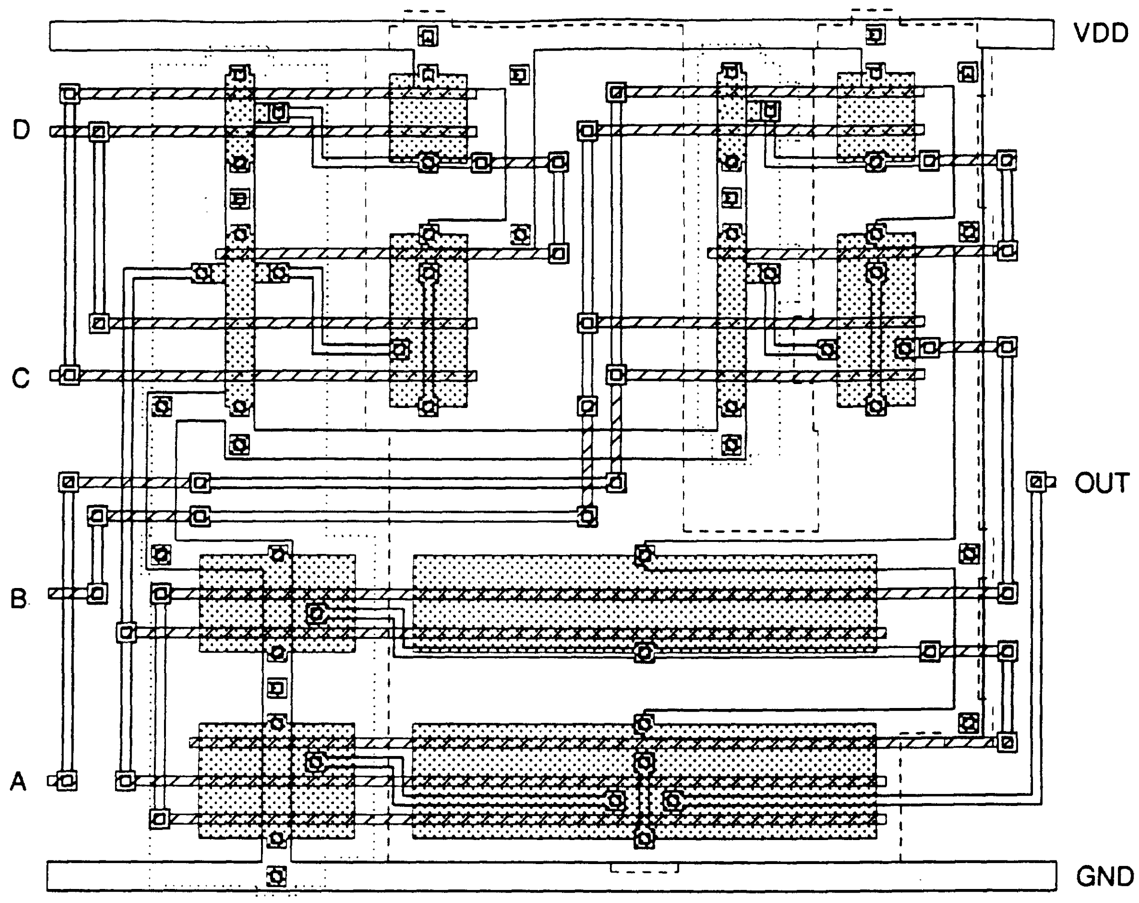

Figure 11-14: Four Input Exclusive-Or Layout

11.1.7. A Voting Circuit

This example illustrates the application of SOLO to a somewhat larger circuit. The circuit shown in this example is used to select an output based on three data inputs and three *mask* inputs (one corresponding to each data line). The output goes low if at least two of the inverted inputs are low and are not masked or if one of the inverted inputs is low and the other two are masked. The schematic for this circuit is given in figure 11-15. The layout is shown in figure 11-16.

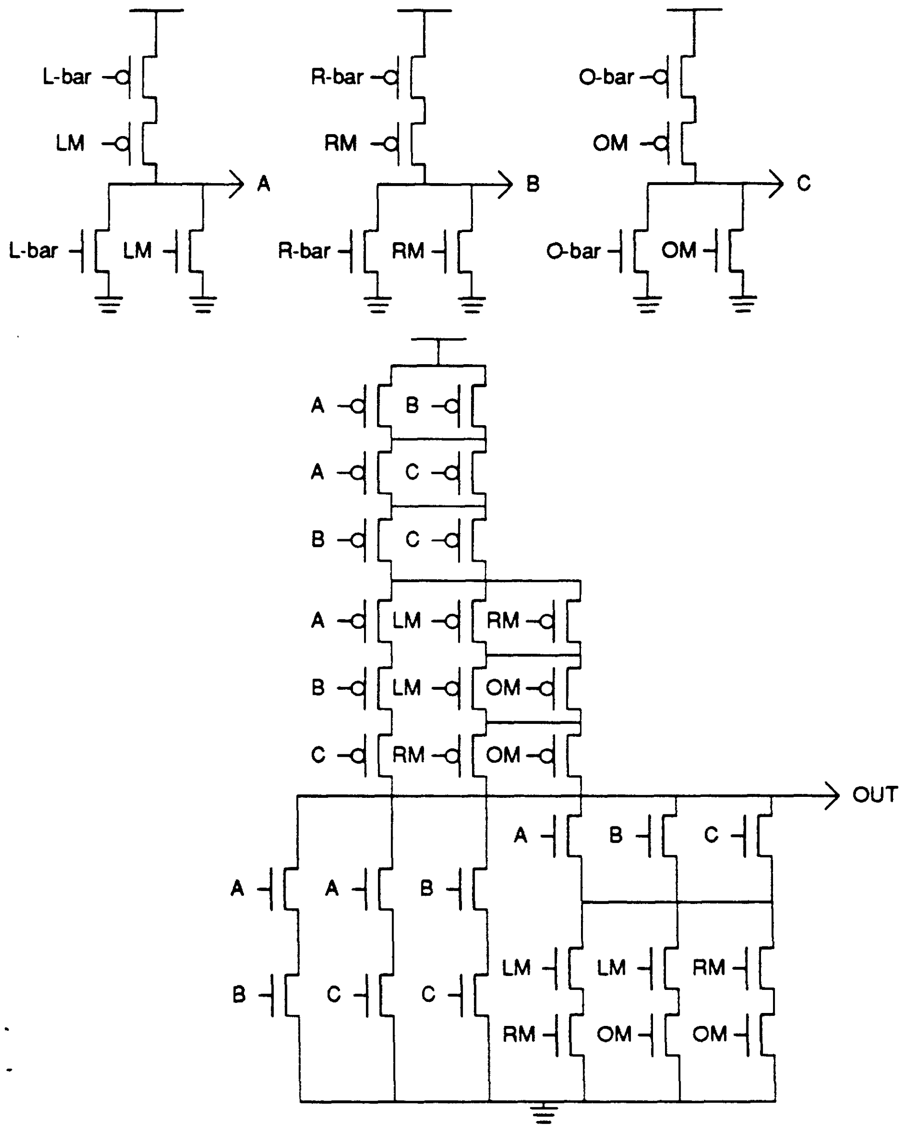

Figure 11-15: A Voting Circuit

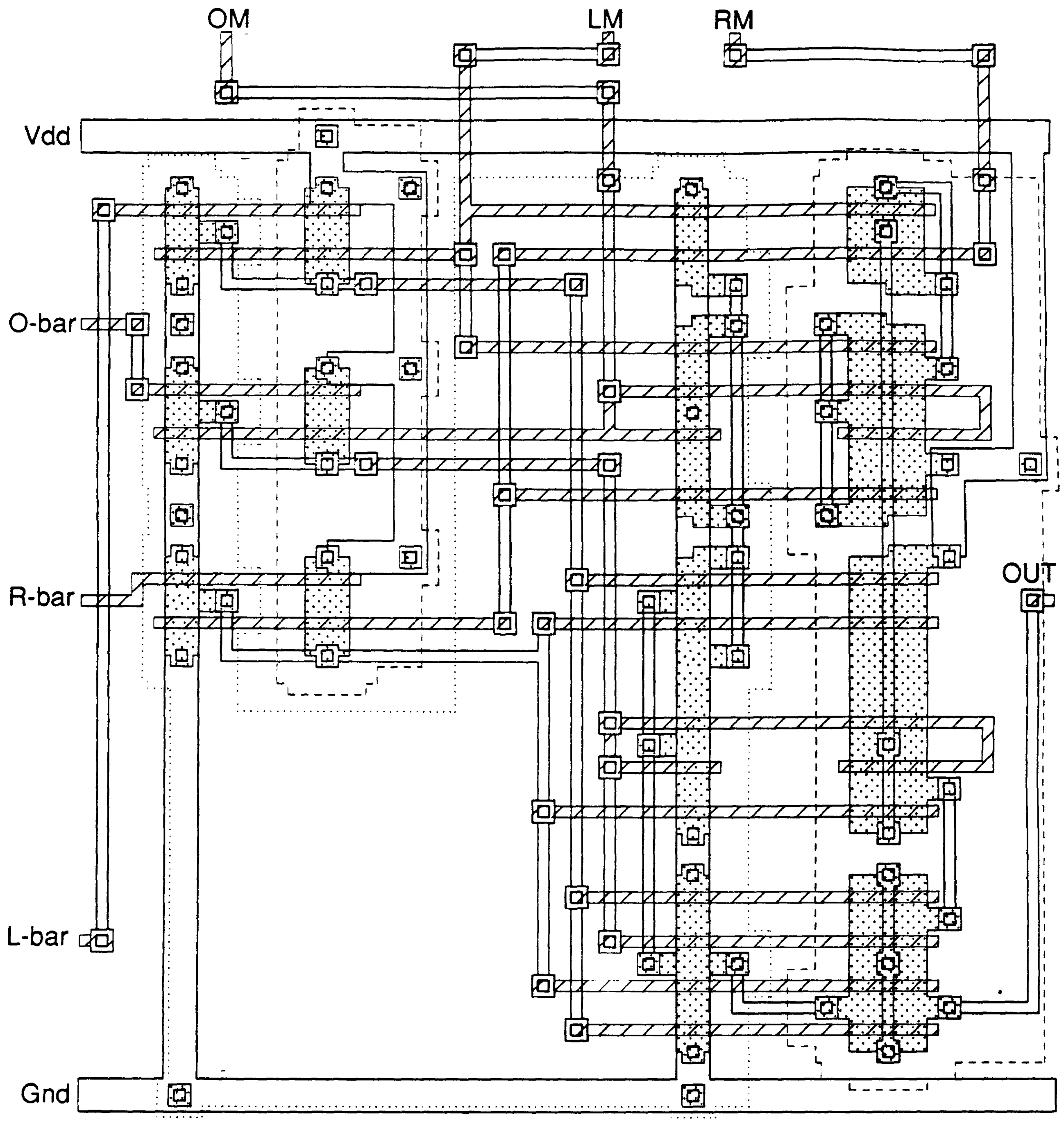

Figure 11-16: Voting Circuit Layout

11.1.8. Four-bit Lookahead Adder Circuit

In this example, SOLO is used to realize a static CMOS implementation of a four-bit adder with carry lookahead circuitry. The circuit includes 148 transistors. The general form of the circuit is shown below. The details of the circuit are given in the circuit specification in appendix B. The generated layout is shown in figure 11-18.

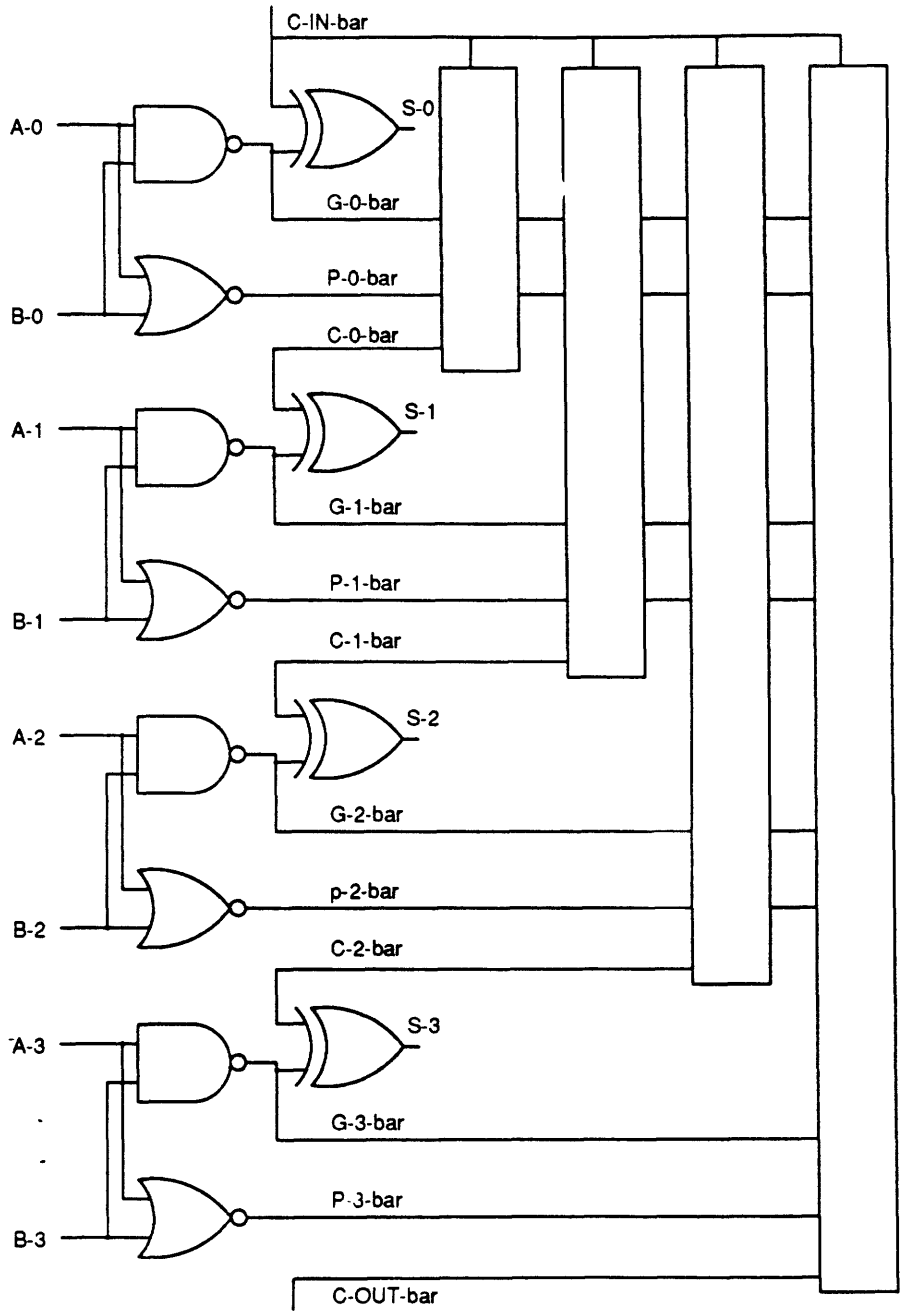

Figure 11-17: Four-bit Lookahead Adder Circuit

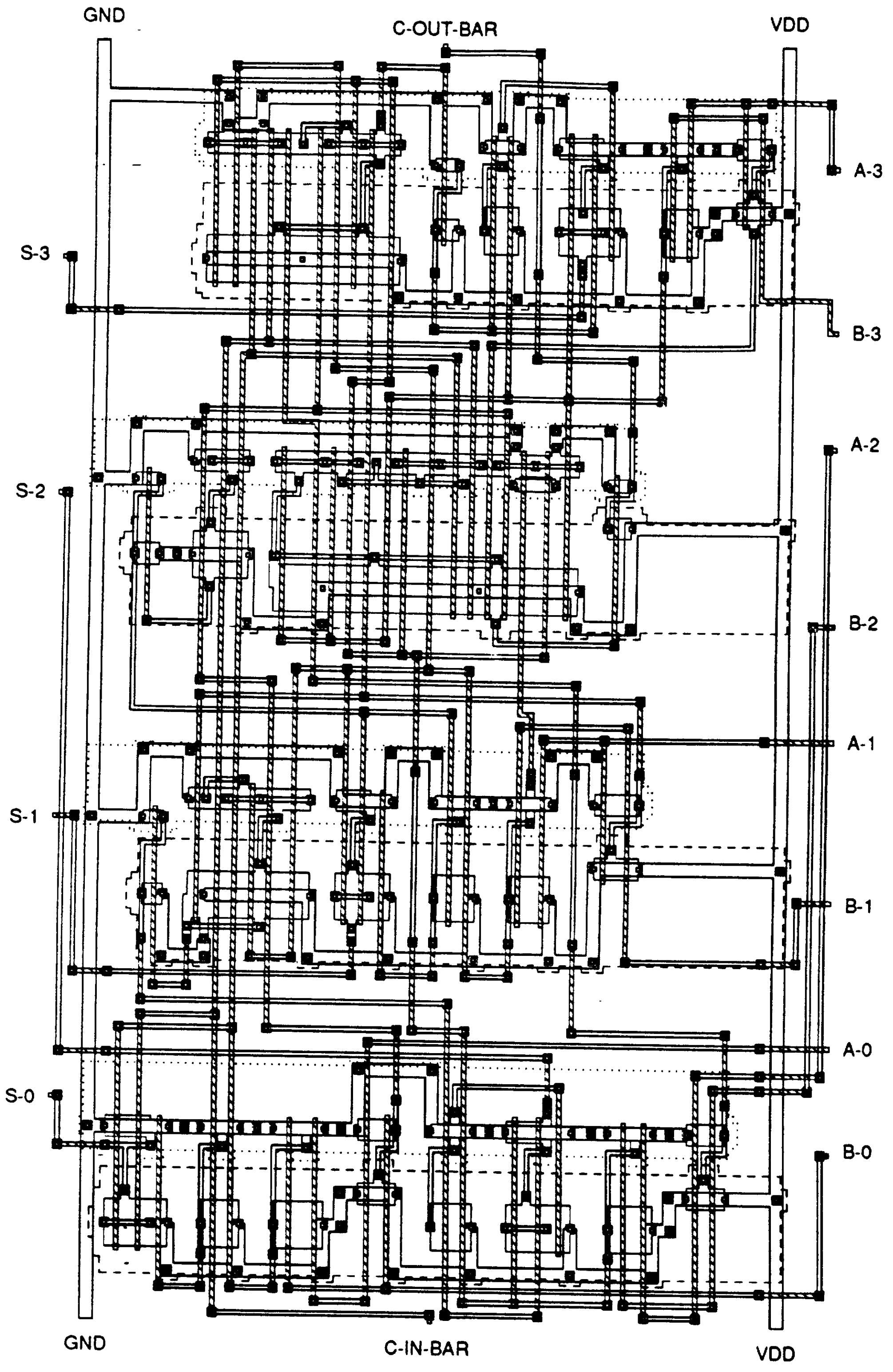

Figure 11-18: Four-bit Lookahead Adder Layout

11.1.9. NMOS Adder Circuit

SOLO can also be applied to NMOS circuits. This example illustrates the realization of an NMOS adder circuit.

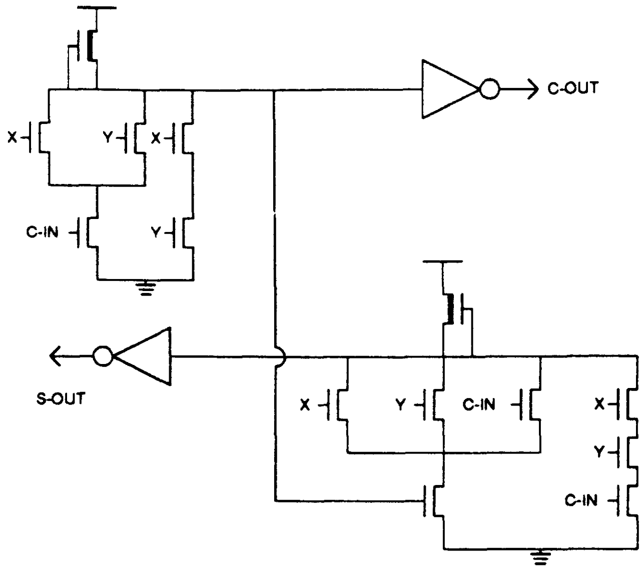

Figure 11-19: NMOS Adder Circuit

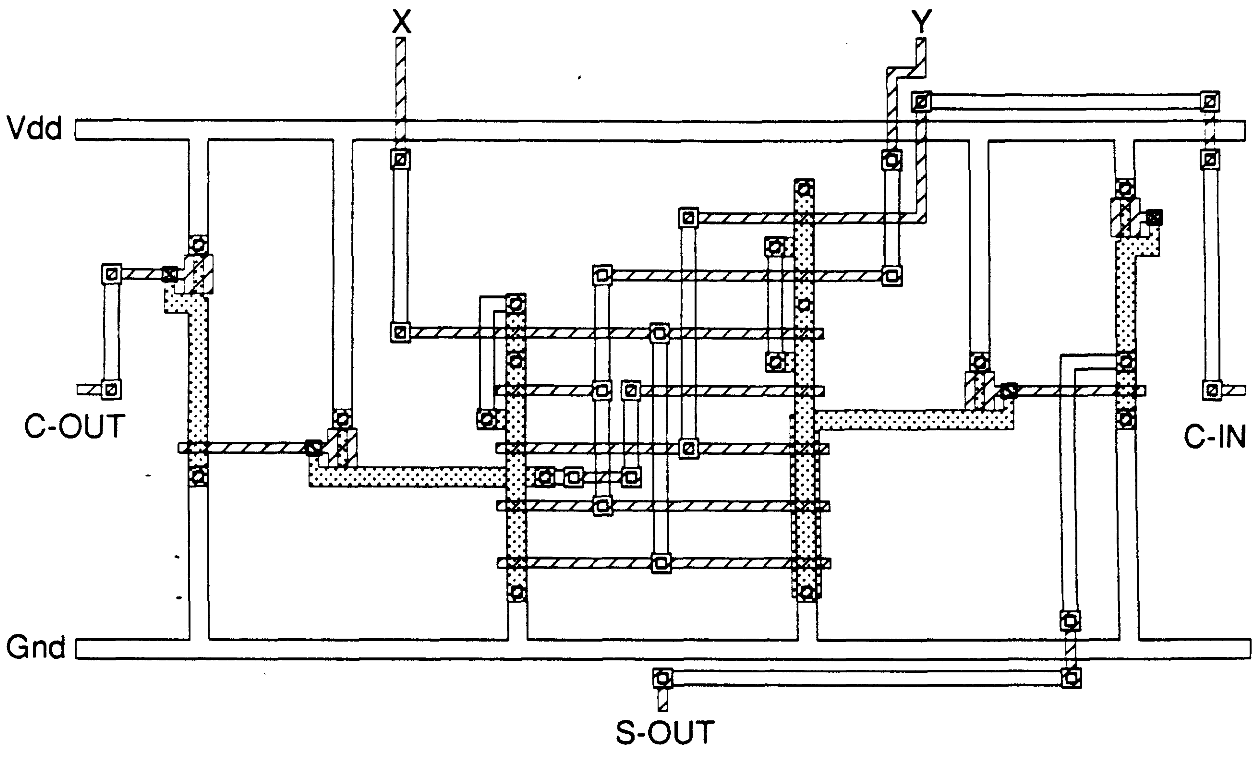

Figure 11-20: NMOS Adder Symbolic Layout

11.2. Analysis

In this section, a critical analysis of SOLO is presented. This analysis discusses the performance of the program as demonstrated by these examples in terms of the original design goals of the research.

11.2.1. Generality

One goal of the research was to design a synthesis program that would be as general in its applicability as possible. It was hoped that the system could effectively be applied to sized transistor circuits of varying methodologies and to circuits of both NMOS and CMOS technologies. It was further hoped that the program would be capable of generating layouts compatible with specific surrounding environments.

The generality of the program in each of these areas has been demonstrated.

- SOLO can be effectively applied to circuits employing many different circuit methodologies. The circuits used in these examples include sized transistors and demonstrate the use of a variety of circuit methodologies including methodologies which do not include equal numbers of n-type and p-type transistors.
- The program can be applied to both NMOS and CMOS circuits.
- Through the use of realization constraints, SOLO can generate layouts that are compatible with specific surrounding environments. These examples made use of desired aspect ratio specifications, of desired connection location specifications, and of feedthrough net specifications.

The degree of generality supported by SOLO in each of these areas equals or exceeds that supported by any other known synthesis program.

11.2.2. Computational Efficiency

Another design goal was to create a program that would run for no longer than a few CPU minutes for a circuit including roughly 20 transistors. It was hoped that the program could be competitive with TALIB in terms of computational efficiency and more efficient than DUMBO, Koss's program or TOPOLOGIZER. This level of computational efficiency would allow the program to be applied to circuits including on the order of 200 transistors while still running in a reasonable amount of time.

This level of computational efficiency has been achieved. A summary of the computation time required for each of these synthesis examples is given in table 11-1. SOLO is implemented in Common Lisp and runs on a Symbolics 3640 Lisp Machine.

Summary of Computation Times			
Circuit	Number of Transistors	Number of Sublayouts	CPU Time (Minutes)
D Flip-Flop	6	6	0.8
Clocked XOR	9	4	0.5
Static XOR	10	4	0.6
NMOS Adder	18	8	1.5
Pass Gate Adder	24	24	7.0
Classical CMOS Adder	28	8	2.0
4-Input XOR	30	12	2.6
Voter Circuit	42	8	4.0
Lookahead Adder	148	48	39.5

Table 11-1: Summary of Computation Times

These computation times are clearly superior to those for DUMBO and Koss's program. Those programs took in excess of twenty minutes for circuits on the order of twenty-five transistors. The ability of SOLO to outperform these programs computationally can be directly attributed to the use of hierarchy. The program uses the existing hierarchy and the new hierarchy (which is introduced during the restructuring phase) to reduce the complexity of the computationally expensive placement phase. The fact that the required computation time is more a function of the number of sublayouts than it is of the number of transistors is indicative of this use of hierarchy.

The computation times of SOLO are also competitive with those of TALIB. Although no exact execution times for TALIB were published, these times seem comparable to those alluded to.

The program is not computationally competitive with either Sc2 or with the IBM program however. The restrictive layout methodologies used by these programs allow them to run at least an order of magnitude faster than SOLO on circuits including large numbers of transistors. Although a reimplementation of SOLO in a more efficient programming language would close this computational gap somewhat, the general nature of the program's layout approach makes it inherently more computationally expensive than the methods employed by Sc2 and the IBM program. This general approach does, however, lead to the generation of higher quality layouts in many design situations.

11.2.3. Layout Efficiency

In terms of layout efficiency, the design goal was to create a program which would generate layouts that were comparable in quality to those generated by TALIB but which was applicable to CMOS circuits.

SOLO is clearly applicable to CMOS circuits and the layouts it generates do seem competitive with those generated by TALIB. Because the compactor presently being used by SOLO only allows the generation of CMOS mask layouts, and because TALIB can only be applied to NMOS circuits, it is not possible to directly compare different realizations of the same circuit as generated by SOLO and TALIB. Although layout quality is difficult to quantify under such conditions, by surveying a number of layouts generated by a synthesis program it is possible to gauge fairly accurately the level of layout expertise that program has attained. In terms of such a *layout expertise* metric, the layouts presented here are comparable with those of TALIB.

Comparing the quality of the layouts generated by SOLO to that of those generated by Sc2 or by the IBM program is also not straightforward. For most circuits, the layouts generated by SOLO are superior to those generated using more restrictive layout methodologies. With these circuits, the circuit partitioning phase is effective and thus many of the same local layout optimizations typical of circuits realized using a gate-matrix methodology can also be incorporated into the layouts generated by SOLO. At the same time, the general placement methodology used by SOLO allows it to be effectively applied to a wider class of circuits. For circuits which do not partition effectively however, the rigid layout approach employed by Sc2 and the IBM program is sometimes more effective in generating quality layouts. Circuits which are of a form that can be effectively realized using a gate matrix technique (and which do not partition effectively using the series-parallel partitioning algorithm) can typically be more effectively realized using the restrictive layout methodologies employed by Sc2 and the IBM program. In these specific cases, the local layout optimizations made possible by the gate matrix technique are of greater importance than the advantages associated with SOLO's more general layout approach. The pass transistor adder (shown in figures 11-9 and 11-10) is an example of a circuit which does not partition effectively and which can be more effectively realized using a more restrictive layout methodology.

Although no human layouts corresponding to the examples presented in this chapter have been created, some general observations can be made regarding the relative quality of these layouts as compared to those created by human designers.

The highest quality layouts are generated when the circuit partitioning is most effective. The classical CMOS adder, the voter circuit, and the lookahead circuit are all examples in which the circuit partitioning was effective and in which quality layouts were generated as a result. These layouts seem only slightly less dense than comparable human designs. The area penalties over human designs in these cases are well below 50%[10], and much of the added area can be attributed simply to the use of symbolic layout and the associated virtual grid compaction scheme. The use of a more effective graph-based compaction scheme would further reduce the area penalty.

For circuits which do not partition effectively, the area penalty over human designs can be much more substantial. In these circuits, the ineffective partitioning prevents many local layout optimizations from being incorporated into the layout. The realization of the pass transistor adder circuit is the worst example of this phenomenon encountered thus far. The generated layout in this case is roughly four times the size of a corresponding human layout.

This analysis of the program's layout efficiency makes it clear that two specific aspects of the synthesis strategy are of primary importance in the generation of quality layouts.

One such aspect is the circuit partitioning phase and the subsequent mapping of the subcircuits into layout using a gate matrix technique. The partitioning and mapping into layout allows local layout optimizations such as connection by abutment to be incorporated into the layout. The importance of these optimizations is illustrated by the

[10]Although an area penalty of 50% may seem quite high, it corresponds to a linear size penalty of only 20% on each side.

fact that the highest quality layouts are generated when the circuit partitioning is most effective.

Another important aspect of SOLO in terms of layout efficiency is the general placement methodology. Although general placement methodologies have been used by all the synthesis programs most successful in generating high quality layouts, general methodologies have never previously been applied to CMOS circuits. The general placement methodology allows non-standard circuit methodologies such as that employed in the clocked exclusive-or circuit to be realized effectively.

11.3. Problems

While the preceding discussion does highlight many of the strengths of the layout approach developed here, some major problems with the present strategy are also revealed. These problems are discussed in more detail below and some possible solutions are proposed.

11.3.1. Circuit Partitioning

One problem with the present approach relates to the effectiveness of the circuit partitioning strategy when applied to certain types of circuits. Although the series-parallel partitioning strategy works quite well for most circuits, some circuits, such as those including many pass transistors, are partitioned quite inefficiently using this technique. The ineffective partitioning prevents many local layout optimizations from being effected and thus leads to inefficient layouts.

The obvious solution to this problem is to develop a new partitioning strategy that can effectively be applied to more classes of circuits. The present strategy was selected because it provides a technique for partitioning a circuit into closely connected subcircuits while at the same time ensuring that the generated subcircuits can be mapped into layout in a straightforward manner using a gate matrix technique. Any new strategy must thus address the problem of circuit partitioning as well as the problem of mapping the more general subcircuits into gate matrix layout.

The algorithm used by the IBM program for mapping circuits onto a gate matrix transistor array [11] provides a viable method for mapping more general subcircuits into gate matrix layout. While no comparable solution to the associated problem of circuit partitioning is immediately known, it is clear that some better solution to the partitioning problem can be found. The improved layout efficiency associated with being able to more effectively partition more classes of circuits certainly makes the search for such a new strategy worthwhile.

11.3.2. Computational Efficiency

Another more subtle difficulty with the program relates to the issue of computational efficiency. Although SOLO's level of computational efficiency is acceptable, it would certainly be desirable for the program to run faster in general and in particular when applied to larger circuits.

As was alluded to earlier, the computational performance of SOLO could be improved a great deal simply by its reimplementation in a different programming language. It is reasonable to expect that the program's execution speed would improve by as much as a factor of ten if the program were reimplemented in a more efficient programming language such as C. The computational performance of SOLO could also be improved through the reimplementation of certain timing critical parts of the program using more efficient algorithms. When applied to very large circuits, the computation time of the present program is dominated by the global routing phase. This part of the program is quite computationally expensive because it is applied to a flattened layout description. The global router does not take advantage of the hierarchical nature of the binary tree layout description. If the global routing algorithm could be modified so as to somehow be applied in a hierarchical fashion, the computational efficiency of the program as a whole could be greatly improved.

Chapter 12
Conclusions

12.1. Summary

In developing large integrated systems, circuit designers have typically been forced to choose between the rapid design times afforded by automated layout systems and the high performance layouts generally associated with full custom designs. The goal of this research was to develop an automated layout methodology through which the performance penalties generally associated with automated layout systems could be avoided.

A program for mapping MOS circuit schematics into layout was developed to provide a vehicle through which this goal could be pursued. The resultant program can effectively be applied to circuits employing many different circuit methodologies, to circuits including sized transistors, and to circuits employing either NMOS or CMOS technologies. The computational efficiency of the program is such that circuits including up to roughly 200 transistors may be realized. The layouts generated are superior in performance to those generated by conventional automated layout systems and for some classes of circuits are competitive with layouts created by human designers.

12.2. Overall Layout Strategy

The strategy for automated layout demonstrated here involves the application of general techniques most often applied on a global level (with standard cells for instance) to the more local problem of layout synthesis from circuit schematics. A general layout technique was employed because general strategies allow the widest variety of circuit methodologies to be effectively realized. This has been demonstrated by the success of NMOS layout synthesis programs which employ general layout strategies.

The general approach to automated layout has a number of associated problems however.

1. General layout strategies do not typically effect local layout optimizations.
2. General layout strategies are difficult to apply to CMOS layout.

A hybrid layout strategy was thus developed which preserves many of the desirable aspects of a general strategy while at the same time minimizing the problems outlined above.

A general strategy is only applied to the non-leaf (more global) levels of the hierarchy. These levels tend to be modular (with a minimum number of interconnections between modules) thus local layout optimizations are rarely appropriate on these levels even in an optimal layout. A more restrictive gate matrix layout methodology is applied to the leaf subcircuits. This methodology allows many local layout optimizations to be incorporated into the generated layout. Although the restrictive gate matrix methodology can only effectively be applied to specific types of circuits, the combined strategy can be applied to circuits of any type. Since the gate matrix strategy is only applied to the leaf subcircuits, the specific nature of the circuit as a whole is not restricted.

In order to improve the effectiveness of this mixed strategy, an initial restructuring of the circuit description hierarchy is included. This restructuring results in a more modular description hierarchy (in the layout domain), and leads to leaf subcircuits which can be effectively realized using the target gate matrix methodology.

The employed strategy also addresses the problem of applying general layout techniques to CMOS circuits. The issues that make the application of general strategies to CMOS layout difficult are well and substrate contact placement. In order to ease well placement, a special suitability for well positioning scoring function is included in the general placement phase. In addition, the global routing program is designed to ensure that substrate contacts can be easily positioned in appropriate locations. By addressing the issues of well and substrate contact placement at a high level, their subsequent positioning becomes trivial. The consideration of CMOS-specific layout issues on a global level allows general layout techniques to effectively be applied to CMOS circuits.

Finally, in order to further enhance the effectiveness of general techniques when applied to layout problems that are typically quite local in nature, scoring functions are used which incorporate both local and global layout issues. The employed electrical performance scoring function includes two factors, one biased towards local layout considerations and the other towards more global considerations. Similarly, the suitability for well positioning scoring function addresses both local and global layout issues. On a local level, the function tends to orient sublayouts so as to ease their connection to power and ground lines. On a global level, layouts of the same diffusion type tend to be clustered.

12.3. Perspective

The ultimate goal of research in this area is to develop a program that can consistently generate layouts that are competitive with those created by human designers. Although it is clear that SOLO in itself does not represent the attainment of this ultimate goal, the lessons learned through SOLO's development do provide some valuable insights which will aid in the quest for that final goal.

The success of SOLO in generating high quality layouts in certain design situations demonstrates the general usefulness of several aspects of the developed layout strategy. A program that is to generate layouts that are consistently competitive with human layouts should incorporate these effective aspects of SOLO's layout strategy.

Like SOLO, an effective layout synthesis program should be both flexible and controllable. The *optimal* layout of a given circuit will be different in different design situations. In order to compete with human designers an automated layout program must be able to adjust to different surrounding environments and design requirements.

It further seems clear that the program should employ a mixed layout strategy. It should deal differently with local layout issues than it does with global layout issues.

- On a local level, the basic strategy should be layout optimization. This optimization should include the removal of unnecessary wires, contacts, and jogs and the realization of connections by abutment. In order to allow connections by abutment to be made, a restrictive layout methodology should be applied at the leaf level. SOLO effects these local optimizations through a specific layout optimization phase in which unnecessary wires, contacts, and layer changes are removed and through the application of a gate matrix layout methodology to the leaf subcircuits.
- On a global level, general layout techniques should be employed. Above the leaf level, general techniques introduce few costs in the way of layout area or electrical performance. These techniques will allow the program to

effectively be applied to many different circuit methodologies. In addition, the scoring functions typically used by general layout approaches provide a convenient means for representing and incorporating application specific knowledge. The scoring functions used by SOLO incorporate CMOS-specific knowledge, router-specific knowledge, and address both local and global layout issues.

In order for the strategy described above to be effective, a restructuring of the circuit description hierarchy should also be included. The restructuring should ensure that the description hierarchy is modular in the layout domain and that the leaf subcircuits can effectively be realized using the target restrictive layout methodology.

An analysis of the factors involved in SOLO's inability to generate quality layouts for certain types of circuits reveals a number of improvements that should also be included in any subsequent programs that are to be truly competitive with human designers.

The most obvious improvement that is needed is a better restructuring strategy. The new strategy should lead to the effective partitioning of essentially all classes of circuits. The only layouts generated by SOLO that were of significantly lesser quality than those generated by human designers were those in which the partitioning strategy was ineffective. The use of a partitioning strategy that could effectively be applied to nearly all classes of circuits would lead to the generation of more efficient layouts much closer in quality to those generated by human designers.

A more subtle needed improvement involves the effective use of layout replication in appropriate situations. In order to avoid the downfalls of more conventional automated layout systems (such as standard cells), in which each instance of a given subcircuit is realized using an identical layout, SOLO was designed to adjust the layout realization of each instance of a given subcircuit to its specific environment. In some design situations however (such as with bit-slice designs for instance), both computational efficiency and layout efficiency could be improved if one appropriate layout could be generated and then replicated many times. It is the detection of *appropriate* design situations that makes this problem difficult. When should an instance be replicated and when should it be adjusted to the specific surrounding environment? The ability to detect and take advantage of such replication situations would allow higher quality layouts to be generated while at the same time improving computational efficiency.

A final improvement that is needed relates to computational efficiency. In order for a synthesis program to be competitive with human designers its applicability must not be limited to circuits with on the order of 200 transistors (as SOLO presently is). The key to increasing the size of circuit to which a program can effectively be applied involves increasing that program's computational efficiency. A truly competitive layout synthesis program must be more computationally efficient than SOLO. The needed improvement in computational speed could be effected in a number of ways.

- Algorithmic improvements in the program's implementation could be made.
- The program could be implemented in a more computationally efficient programming language.
- The program could be run on more computationally efficient hardware.

Hopefully, the lessons learned through SOLO's development and outlined above will help bring the next generation of layout synthesis programs closer to the ultimate goal of replacing human layout.

12.4. Future Extensions

Although SOLO can be applied to optimized circuit schematics, its effectiveness in generating high performance layouts is inherently limited by the open loop nature of the synthesis process. By combining the program with a computationally efficient delay estimation program, a closed loop synthesis system could be developed. Information from a layout generated on one pass through the system would be used by a transistor sizing program to generate a new sized circuit schematic for the next pass. By repeating this process several times, layouts which meet specific delay performance constraints could be generated.

Another extension to SOLO would involve its incorporation into a complete silicon compiler system. Recall that mapping a design from the functional domain into the physical domain is a two step transformation. SOLO would provide a means for mapping intermediate structural representations into layout specifications. The associated compiler would thus be free to select the circuit topologies and transistor sizings best suited to specific design situations. An expert system could provide a means by which appropriate realization circuit methodologies would be selected. The layouts generated by such a compiler would be of better quality than those generated by a compiler constrained to use a much smaller topological design space.

Appendix A
Specification Language Syntax

A.1. Introduction

This appendix presents a summary of the circuit specification language syntax.

A.2. Specifying a Design

A.2.1. Defschematic

The macro used to specify a design to be realized is **defschematic**.

(defschematic *name* (*external-nodes*)
  [**(local** *local-nodes*)]
  body)

Each **defschematic** must include a *name* and a list of *external-nodes*. If any *local-nodes* are used in the *body* of the **defschematic**, they must be listed in the first line of the *body* through use of the **local** function. The remainder of the *body* consists of functions and macros which define the topology of the circuit as well as any desired realization constraints.

Any nodes referenced in the defschematic other than **vdd** and **gnd** must be included in either the *local-nodes* or the *external-nodes* lists. **vdd** and **gnd** are global nodes and should not be included in either list.

A.3. Realization Constraints

This section describes the functions used in specifying desired realization constraints which control the exact form of the generated layout.

A.3.1. Technology

The desired realization technology is specified using the **set-technology** function.

(**set-technology** *technology*)

A.3.2. Aspect Ratio

A desired aspect ratio is specified using the **set-aspect-ratio** function.

(**set-aspect-ratio** *width height*)

This function takes a *width* and a *height* as arguments. These both must be integers greater than 0. If no desired aspect ratio is specified, a desired aspect ratio of 1 to 1 is assumed.

A.3.3. Net Weighting

The relative weightings of nets can be set using the **set-weighting** function.

(**set-weighting** *node value*)

This function takes a *node* and a *value* as arguments. *Value* must be an integer between 0 and 50. The relative weighting of the specified *node* is set to *value*. The default weighting for all signal nodes is 10.

A.3.4. Connection Locations

The desired locations of all external signal connections must be specified. Desired locations are specified using the **top**, **bottom**, **left**, and **right** functions.

(**top** *nodes keyword-value-pairs*)

(bottom *nodes keyword-value-pairs***)**

(left *nodes keyword-value-pairs***)**

(right *nodes keyword-value-pairs***)**

Each of these functions takes a list of *nodes* as arguments as well as an optional **:ordered-p** keyword with an associated value. The connections of the *nodes* included in each argument list are located on the appropriate side. The keyword specifies whether the connection locations must be ordered as they appear in the argument list. **:ordered-p** takes a value of either *t* or *nil* and has a default value of *t*. If they are ordered, **top** and **bottom** connections are ordered from left to right while **left** and **right** connections are ordered from bottom to top.

Each of these functions can be called only once and each external node of the circuit must be included in the argument list of at least one function.[11] If an external node appears more than once, multiple external connections for the same node will be generated. It is assumed that these multiple connections are externally connected. This assumption can be changed by using the **not-externally-connected** function.

(not-externally-connected *nodes***)**

This function takes a list of *nodes* as arguments. It will not be assumed that multiple external connections of these *nodes* are externally connected.

By using the **horizontal-align** and **vertical-align** functions it is also possible to constrain connections such that they will be aligned (in one dimension) with appropriate connections on the opposite side of the cell. In this way cells can be generated that can be placed in an array and connected by abutment.

(horizontal-align)

(vertical-align)

[11]This is not entirely true. Feedthrough connections, **vdd**, and **gnd** are not included in these argument lists.

The **horizontal-align** function aligns connections on the left and right sides of the cell while the **vertical-align** function aligns connections on the top and bottom.

A.3.5. Feedthroughs

Feedthrough nets can be specified using the **top-feed-through** and **bottom-feed-through** functions.

(top-feed-through *nodes***)**

(bottom-feed-through *nodes***)**

Each of these functions specifies two connection locations for each node, one on the left side and one on the right side of the cell. The two connections are connected by a metal feedthrough and are aligned with each other in the Y dimension. Feedthroughs specified with the **top-feed-through** function are positioned near the top of the cell above the VDD feedthrough. Those specified with the **bottom-feedthrough** function are positioned near the bottom of the cell below the GND feedthrough. Feedthrough connections should not be included in the **left** or **right** connection location specifications unless extra connections (in addition to the feedthrough connections) are desired.

A.4. Specifying Topology

This section describes the syntax used in defining the desired circuit topology.

A.4.1. Circuit Branch Descriptors

Circuit branch descriptors are the basic functions used to specify topology. Functions exist for specifying individual transistors, for specifying series combinations of branches, and for specifying parallel combinations of branches.

A.4.1.1. Transistors

Transistors are specified using the **transistor** function. This function takes a *gate-node* as an argument as well as optionally a *drain-node* and a *source-node*.

(transistor [*drain-node source-node*] (*gate-node*)
    *keyword-value-pairs***)**

If this function is invoked at the top level, the *drain-node* and *source-node* must be included. If it is called within a series or a parallel branch descriptor, the *drain-node* and *source-node* must be omitted.

A number of keyword value pairs may also be used to specify the exact characteristics of the transistor.

:type	This keyword is used to specify the type of transistor. Possible values are *n*, *p*, *d*, and *z*. It should be noted that only a subset of these are applicable to any given realization technology. The default value is *n*.
:width	This keyword is used to specify the width of the transistor in units of lambda. The default value is 2.
:length	This keyword is used to specify the length of the transistor in units of lambda. The default value is 2.

A.4.1.2. Parallel Branches

Parallel connections of transistors (or of other branches) are specified using the **parallel** function. This function takes a list of *gate-nodes-or-branches* as an argument as well as optionally two connecting nodes (*node-1* and *node-2*).

(parallel [*node-1 node-2*] (*gate-nodes-or-branches*)
    keyword-value-pairs)

If this function is invoked at the top level, the *node-1* and *node-2* arguments must be included. If it is called within a series or a parallel branch descriptor, these arguments must be omitted.

If the **parallel** function is specifying the parallel combination of a set of transistors all of which have identical characteristics, then the transistors may be specified simply by their associated gate nodes. The characteristics of those transistors are specified using the same keywords as were used with the **transistor** function.

If the function is specifying the parallel combination of more complex branches or of transistors with different characteristics, then each branch must be explicitly specified. The **:type**, **:width**, and **:length** keywords are no longer applicable.

A.4.1.3. Series Branches

Series connections of transistors (or of other branches) are specified using the **series** function.

(**series** [*node-1 node-2*] (*gate-nodes-or-branches*)
keyword-value-pairs)

The use of this function is identical to that of the **parallel** function with one exception. The **series** function can be invoked with one more possible keyword value pair.

:ordered-p	This keyword specifies whether the specified ordering of the branches or transistors connected in series is relevant. The possible values are *t* and *nil*. The default value is *nil*.

A.4.2. Circuit Macros

A **circuit-macro** is used to specify the topology of a larger subcircuit.

(**circuit-macro** *name* (*external-nodes*)
[(**local** *local-nodes*)]
body)

Each **circuit-macro** must include a *name* and a list of *external-nodes*. If any *local-nodes* are used in the *body* of the **circuit-macro**, they must be listed in the first line of the *body* through use of the **local** function. Once again, the **vdd** and **gnd** nodes should not be included in either list. The remainder of the *body* consists of functions and macros which define the topology of the subcircuit. Functions specifying realization constraints are generally not included in a **circuit-macro** since these functions will affect the realization of the top level **defschematic**.

A **circuit-macro** must be defined before it is invoked. A **circuit-macro** is called like any other function. The syntax is shown below.

(*macro-name nodes*)

A.5. An Example Defschematic

In this section a sample **defschematic** illustrating many of the functions and constructs described here is presented.

```
(circuit-macro pull-up (phi output)
               (transistor vdd output
                           (phi) :type p :width 15 :length 3))

(circuit-macro pull-down (phi output)
               (transistor output gnd
                           (phi) :type n :width 15 :length 3))

(defschematic clocked-xor (a b out phi-1 phi-2)
  (local x y z)
  (set-technology cmos21)
  (left a b)
  (right out)
  (top-feed-through phi-1 phi-2)
  (bottom-feed-through phi-1 phi-2)
  (pull-up phi-1 x)
  (parallel x y
          (a b) :type n :width 15 :length 3)
  (pull-down phi-1 y)
  (pull-up phi-2 out)
  (parallel out z
          ((transistor (x) :type n :width 15 :length 3)
           (series (a b) :type n :width 15 :length 3)))
  (pull-down phi-2 z))
```

Appendix B
Specification Language Descriptions

B.1. Introduction

In this appendix the specification language descriptions of the example circuits presented in chapter 11 are given.

B.2. Exclusive-Or Circuit

```
(circuit-macro nor (a b out)
               (series vdd out (a b)
                       :type p :width 24 :length 3)
               (parallel out gnd (a b)
                         :type n :width 8 :length 3))


(defschematic xor (a b out)
  (local c)
  (set-technology cmos21)
  (left a b)
  (right out)
  (nor a b c)
  (series vdd out
          ((transistor (c) :type p :width 24 :length 3)
           (parallel (a b) :type p :width 24 :length 3)))
  (series out gnd (a b) :type n :width 16 :length 3)
  (transistor out gnd (c) :type n :width 8 :length 3))
```

B.3. Clocked Exclusive-Or Circuit

```
(circuit-macro pull-up (phi output)
               (transistor vdd output
                           (phi) :type p :width 15 :length 3))

(circuit-macro pull-down (phi output)
               (transistor output gnd
                           (phi) :type n :width 15 :length 3))

(defschematic clocked-xor (a b out phi-1 phi-2)
  (local x y z)
  (set-technology cmos21)
  (left a b)
  (right out)
  (top-feed-through phi-1 phi-2)
  (bottom-feed-through phi-1 phi-2)
  (pull-up phi-1 x)
  (parallel x y
           (a b) :type n :width 15 :length 3)
  (pull-down phi-1 y)
  (pull-up phi-2 out)
  (parallel out z
           ((transistor (x) :type n :width 15 :length 3)
            (series (a b) :type n :width 15 :length 3)))
  (pull-down phi-2 z))
```

B.4. Classical CMOS Adder Circuit

```
(circuit-macro inverter (input output)
               (transistor vdd output
                           (input) :type p :width 12 :length 3)
               (transistor output gnd
                           (input) :type n :width 4 :length 3))



(defschematic classic-full-adder-1 (x y c-in c-out s-out)
  (local s-out-bar c-out-bar)
  (set-technology cmos21)
  (top x y)
  (bottom s-out)
  (right c-out)
  (left c-in)
  (horizontal-align)
  (parallel c-out-bar gnd
            ((series
               ((parallel (x y) :type n :width 4 :length 3)
                (transistor (c-in) :type n :width 4 :length 3)))
             (series (x y) :type n :width 4 :length 3)))
  (series vdd c-out-bar
          ((parallel
             ((series (x y) :type p :width 18 :length 3)
              (transistor (c-in) :type p :width 9 :length 3)))
           (parallel (x y) :type p :width 18 :length 3)))
  (parallel s-out-bar gnd
            ((series
               ((parallel (x y c-in) :type n :width 4 :length 3)
                (transistor (c-out-bar)
                            :type n :width 4 :length 3)))
             (series (x y c-in) :type n :width 6 :length 3)))
  (series vdd s-out-bar
          ((parallel
             ((series (x y c-in) :type p :width 24 :length 3)
              (series (c-out-bar) :type p :width 8 :length 3)))
           (parallel (x y c-in) :type p :width 24 :length 3)))
  (inverter c-out-bar c-out)
  (inverter s-out-bar s-out))
```

B.5. Pass Transistor Adder Circuit

```
(circuit-macro inverter (input output)
               (transistor vdd output
                           (input) :type p :width 12 :length 3)
               (transistor output gnd
                           (input) :type n :width 8 :length 3))

(circuit-macro pass-transistor (input output phi phi-bar)
               (transistor input output
                           (phi-bar) :type p :width 12 :length 3)
               (transistor input output
                           (phi) :type n :width 8 :length 3))

(defschematic pass-adder (a b c c-out s-out)
  (local a-bar c-bar a+b a+b-bar s-out-bar c-out-bar)
  (set-technology cmos21)
  (top a b)
  (bottom s-out)
  (left c-out)
  (right c)
  (horizontal-align)
  (transistor a-bar a+b-bar
              (b) :type p :width 12 :length 3)
  (pass-transistor b a+b-bar a a-bar)
  (pass-transistor a+b b a-bar a)
  (transistor a+b a
              (b) :type p :width 12 :length 3)
  (pass-transistor a-bar c-out-bar a+b-bar a+b)
  (pass-transistor c-out-bar c-bar a+b a+b-bar)
  (pass-transistor c-bar s-out-bar a+b-bar a+b)
  (pass-transistor s-out-bar c a+b a+b-bar)
  (transistor a-bar a+b
              (b) :type n :width 8 :length 3)
  (transistor a+b-bar a
              (b) :type n :width 8 :length 3)
  (inverter a a-bar)
  (inverter c c-bar)
  (inverter s-out-bar s-out)
  (inverter c-out-bar c-out))
```

B.6. D Flip-Flop Circuit

```
(circuit-macro inverter (input output)
               (transistor vdd output
                           (input) :type p :width 10 :length 3)
               (transistor output gnd
                           (input) :type n :width 4 :length 3))

(circuit-macro pass-transistor (input output phi phi-bar)
               (transistor input output
                           (phi-bar) :type p :width 20 :length 3)
               (transistor input output
                           (phi) :type n :width 8 :length 3))

(defschematic d-flip-flop (d q q-bar phi phi-bar)
  (set-technology cmos21)
  (top-feed-through phi-bar)
  (bottom-feed-through phi)
  (left d)
  (right q q-bar)
  (pass-transistor d q phi phi-bar)
  (inverter q q-bar)
  (inverter q-bar q))
```

B.7. Four-Input Exclusive-Or Circuit

```
(circuit-macro s-nor (a b out)
               (series vdd out (a b)
                       :type p :width 24 :length 3)
               (parallel out gnd (a b)
                         :type n :width 8 :length 3))

(circuit-macro b-nor (a b out)
               (series vdd out (a b)
                       :type p :width 144 :length 3)
               (parallel out gnd (a b)
                         :type n :width 48 :length 3))

(circuit-macro s-xor (a b out)
               (local c)
               (s-nor a b c)
               (series vdd out
                       ((transistor (c) :type p :width 24 :length 3)
                        (parallel (a b) :type p :width 24 :length 3)))
               (series out gnd (a b) :type n :width 8 :length 3)
               (transistor out gnd (c) :type n :width 8 :length 3))

(circuit-macro b-xor (a b out)
               (local c)
               (b-nor a b c)
               (series vdd out
                       ((transistor (c) :type p :width 144 :length 3)
                        (parallel (a b) :type p :width 144 :length 3)))
               (series out gnd (a b) :type n :width 48 :length 3)
               (transistor out gnd (c) :type n :width 48 :length 3))

(defschematic xor-4 (a b c d out)
  (local e f)
  (set-technology cmos21)
  (left a b c d)
  (right out)
  (s-xor a b e)
  (s-xor c d f)
  (b-xor e f out))
```

B.8. A Voting Circuit

```
(circuit-macro nor (a b out)
              (parallel out gnd
                       (a b) :type n :width 8 :length 3)
              (series vdd out
                     (a b) :type p :width 12 :length 3))

(circuit-macro blob (a b c lm rm om out)
              (parallel out gnd
                       ((series (a b) :type n :width 8 :length 3)
                        (series (a c) :type n :width 8 :length 3)
                        (series (b c) :type n :width 8 :length 3)
                        (series
                          ((parallel (a b c) :type n :width 8 :length 3)
                           (parallel
                             ((series (lm rm) :type n :width 8 :length 3)
                              (series (lm om) :type n :width 8 :length 3)
                              (series (rm om) :type n :width 8 :length 3)))))))
              (series vdd out
                     ((parallel (a b) :type p :width 20 :length 3)
                      (parallel (a c) :type p :width 20 :length 3)
                      (parallel (b c) :type p :width 20 :length 3)
                      (parallel
                        ((series (a b c) :type p :width 20 :length 3)
                         (series
                           ((parallel (lm rm) :type p :width 20 :length 3)
                            (parallel (lm om) :type p :width 20 :length 3)
                            (parallel (rm om) :type p :width 20 :length 3))))))))


(defschematic voting-circuit (l-bar r-bar o-bar lm rm om out)
  (local a b c)
  (set-technology cmos21)
  (top om lm rm)
  (left l-bar r-bar o-bar)
  (right out)
  (nor l-bar lm a)
  (nor r-bar rm b)
  (nor o-bar om c)
  (blob a b c lm rm om out))
```

B.9. Four-bit Lookahead Adder Circuit

```
(circuit-macro inverter (in out)
               (transistor vdd out (in)
                       :type p :width 16 :length 3)
               (transistor out gnd (in)
                         :type n :width 8 :length 3))

(circuit-macro nor (a b out)
               (series vdd out (a b)
                       :type p :width 32 :length 3)
               (parallel out gnd (a b)
                         :type n :width 8 :length 3))

(circuit-macro nand (a b out)
               (parallel vdd out (a b)
                         :type p :width 16 :length 3)
               (series out gnd (a b)
                         :type n :width 16 :length 3))

(circuit-macro x-or (a b out)
               (local c)
               (nor a b c)
               (series vdd out
                       ((transistor (c) :type p :width 32 :length 3)
                        (parallel (a b) :type p :width 32 :length 3)))
               (series out gnd (a b) :type n :width 16 :length 3)
               (transistor out gnd (c) :type n :width 8 :length 3))


(circuit-macro gp (a b g-bar p-bar)
               (nor a b p-bar)
               (nand a b g-bar))

(circuit-macro sum-gen (p-bar c-1-bar sum)
               (x-or p-bar c-1-bar sum))

(circuit-macro bit (a b c-in-bar s g-bar p-bar)
               (gp a b g-bar p-bar)
               (sum-gen p-bar c-in-bar s))


(circuit-macro look-ahead-0 (c-in-bar p-0-bar g-0-bar c-0-bar)
               (local c-0)
               (series c-0 gnd
                         ((parallel (p-0-bar c-in-bar) :type n :width 16 :length 3)
                          (transistor (g-0-bar) :type n :width 16 :length 3)))
               (parallel vdd c-0
                         ((series (p-0-bar c-in-bar) :type p :width 32 :length 3)
                          (transistor (g-0-bar) :type p :width 32 :length 3)))
               (inverter c-0 c-0-bar))

(circuit-macro look-ahead-1 (c-in-bar p-0-bar g-0-bar p-1-bar g-1-bar c-1-bar)
               (local c-1)
               (series c-1 gnd
                       ((parallel (p-1-bar p-0-bar c-in-bar) :type n :width 16 :length 3)
                        (parallel (p-1-bar g-0-bar) :type n :width 16 :length 3)
                        (transistor (g-1-bar) :type n :width 16 :length 3)))
               (parallel vdd c-1
                         ((series (p-1-bar p-0-bar c-in-bar) :type p :width 32 :length 3)
                          (series (p-1-bar g-0-bar) :type p :width 32 :length 3)
                          (transistor (g-1-bar) :type p :width 32 :length 3)))
               (inverter c-1 c-1-bar))

(circuit-macro look-ahead-2 (c-in-bar p-0-bar g-0-bar p-1-bar g-1-bar
                                     p-2-bar g-2-bar c-2-bar)
               (local c-2)
               (series c-2 gnd
                      ((parallel (p-2-bar p-1-bar p-0-bar c-in-bar)
                                 :type n :width 16 :length 3)
                       (parallel (p-2-bar p-1-bar g-0-bar) :type n :width 16 :length 3)
                       (parallel (p-2-bar g-1-bar) :type n :width 16 :length 3)
                       (transistor (g-2-bar) :type n :width 16 :length 3)))
               (parallel vdd c-2
                        ((series (p-2-bar p-1-bar p-0-bar c-in-bar)
                                 :type p :width 32 :length 3)
                         (series (p-2-bar p-1-bar g-0-bar) :type p :width 32 :length 3)
                         (series (p-2-bar g-1-bar) :type p :width 32 :length 3)
                         (transistor (g-2-bar) :type p :width 32 :length 3)))
               (inverter c-2 c-2-bar))

(circuit-macro look-ahead-3 (c-in-bar p-0-bar g-0-bar p-1-bar g-1-bar p-2-bar g-2-bar
                             p-3-bar g-3-bar c-3-bar)
               (local c-3)
               (series c-3 gnd
                      ((parallel (p-3-bar p-2-bar p-1-bar p-0-bar c-in-bar)
                                 :type n :width 16 :length 3)
                       (parallel (p-3-bar p-2-bar p-1-bar g-0-bar)
                                 :type n :width 16 :length 3)
                       (parallel (p-3-bar p-2-bar g-1-bar) :type n :width 16 :length 3)
                       (parallel (p-3-bar g-2-bar) :type n :width 16 :length 3)
                       (transistor (g-3-bar) :type n :width 16 :length 3)))
               (parallel vdd c-3
                        ((series (p-3-bar p-2-bar p-1-bar p-0-bar c-in-bar)
                                 :type p :width 32 :length 3)
                         (series (p-3-bar p-2-bar p-1-bar g-0-bar)
                                 :type p :width 32 :length 3)
                         (series (p-3-bar p-2-bar g-1-bar) :type p :width 32 :length 3)
                         (series (p-3-bar g-2-bar) :type p :width 32 :length 3)
                         (transistor (g-3-bar) :type p :width 32 :length 3)))
               (inverter c-3 c-3-bar))

(defschematic adder (a-0 b-0 a-1 b-1 a-2 b-2 a-3 b-3
                     s-0 s-1 s-2 s-3 c-in-bar c-out-bar)
  (local p-0-bar g-0-bar c-0-bar
         p-1-bar g-1-bar c-1-bar
         p-2-bar g-2-bar c-2-bar
         p-3-bar g-3-bar)
  (set-technology cmos21)
  (top a-3 b-3 a-2 b-2 a-1 b-1 a-0 b-0)
  (bottom s-3 s-2 s-1 s-0)
  (left c-out-bar)
  (right c-in-bar)
  (horizontal-align)
  (bit a-0 b-0 c-in-bar s-0 g-0-bar p-0-bar)
  (bit a-1 b-1 c-0-bar s-1 g-1-bar p-1-bar)
  (bit a-2 b-2 c-1-bar s-2 g-2-bar p-2-bar)
  (bit a-3 b-3 c-2-bar s-3 g-3-bar p-3-bar)
  (look-ahead-0 c-in-bar p-0-bar g-0-bar c-0-bar)
  (look-ahead-1 c-in-bar p-0-bar g-0-bar p-1-bar g-1-bar c-1-bar)
  (look-ahead-2 c-in-bar p-0-bar g-0-bar p-1-bar g-1-bar p-2-bar g-2-bar c-2-bar)
  (look-ahead-3 c-in-bar p-0-bar g-0-bar p-1-bar g-1-bar p-2-bar g-2-bar
                p-3-bar g-3-bar c-out-bar))
```

B.10. NMOS Adder Circuit

```
(circuit-macro pull-up (node)
               (transistor vdd node
                           (node) :type d :width 2 :length 8))


(circuit-macro inverter (input output)
               (pull-up output)
               (transistor output gnd
                           (input) :type n :width 4 :length 2))

(defschematic nmos-full-adder (x y c-in c-out s-out)
  (local c-out-bar s-out-bar)
  (set-technology nmos)
  (top x y)
  (bottom s-out)
  (left c-out)
  (right c-in)
  (horizontal-align)
  (parallel c-out-bar gnd
            ((series
               ((parallel (x y) :type n :width 4 :length 2)
                (transistor (c-in) :type n :width 4 :length 2)))
             (series (x y) :type n :width 4 :length 2)))
  (pull-up c-out-bar)
  (parallel s-out-bar gnd
            ((series
               ((parallel (x y c-in) :type n :width 4 :length 2)
                (transistor (c-out-bar)
                            :type n :width 4 :length 2)))
             (series (x y c-in) :type n :width 6 :length 2)))
  (pull-up s-out-bar)
  (inverter s-out-bar s-out)
  (inverter c-out-bar c-out))
```

References

1. Carver Mead and Lynn Conway, *Introduction to VLSI Systems,* Addison-Wesley Publishing Company, Reading, Massachusetts, VLSI Systems Series, 1980.

2. Neil Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design,* Addison-Wesley Publishing Company, Reading, Massachusetts, VLSI Systems Series, 1985.

3. Arnold Weinberger, "Large Scale Integration of MOS Complex Logic: A Layout Method", *IEEE Journal of Solid-State Circuits,*Vol. SC-2, No. 4, December 1967, pp. 182-190.

4. R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis,* Kluwer Academic Publishers, Boston, The Kluwer International Series in Engineering and Computer Science, 1984.

5. J. M. Siskind, J. R. Southard, and K. W. Crouch, "Generating Custom High Performance VLSI Designs from Succinct Algorithmic Descriptions", *Proceedings, Conference on Advanced Research in VLSI,* Massachusetts Institute of Technology, 1982, pp. 28-39.

6. W. Wolf, J. Newkirk, R. Mathews, and R. Dutton, "Dumbo, A Schematic-to-Layout Compiler", *Proceedings of the Third Caltech Conference on VLSI,* 1983, pp. 379-393.

7. Michael C. Koss, "An Algorithm for Generating VLSI Circuit Topologies", Master's thesis, Massachusetts Institute of Technology, June 1983.

8. Jin H. Kim, *Use of Domain Knowledge in Computer Aid for IC Cell Layout Design,* PhD dissertation, Carnegie-Mellon University, June 1985.

9. Dwight D. Hill, "Sc2: A Hybrid Automatic Layout System", *Proceedings of the International Conference on Computer-Aided Design*, IEEE, 1985, pp. 172-174.

10. S. Wimer, R. Y. Pinter, and J. A. Feldman, "Optimal Chaining of CMOS Transistors in a Functional Cell", *Proceedings of the International Conference on Computer-Aided Design*, IEEE, 1986, pp. 66-69.

11. R. Bar-Yehuda, J. A. Feldman, R. Y. Pinter, and S. Wimer, "Depth First Search and Dynamic Programming Algorithms for Efficient CMOS Cell Generation", Preprint of paper for Proceedings of the Fifth MIT Conference on Advanced Research in VLSI, 1988.

12. Paul W. Kollaritsch and Neil H. E. Weste, "TOPOLOGIZER: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout", *IEEE Journal of Solid-State Circuits,*Vol. SC-20, No. 3, June 1985, pp. 799-804.

13. D. Wells. Private Communication, January 1987

14. Andrei Vladimirescu and Sally Liu, "The Simulation of MOS Integrated Circuits Using SPICE2", Memorandum UCB/ERL M80/7, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, February 1980.

15. T. Uehara and W. M. vanCleemput, "Optimal Layout of CMOS Functional Arrays", *IEEE Transactions on Computers*,Vol. C-30, No. 5, May 1981, pp. 305-312.

16. N. Deo, *Graph Theory with Applications to Engineering and Computer Science,* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1974.

17. R. Nair, A. Bruss, and J. Reif, "Linear Time Algorithms for Optimal CMOS Layout", Research Report RC 10279 (#45821), IBM Thomas J. Watson Research Center, December 1983.

18. R. Rivest, "The PI (placement and routing) System", *Proceedings of the 19th Design Automation Conference*, IEEE, 1982, pp. 475-481.

19. A. T. Sherman, *Cryptology and VLSI (a two-part dissertation): I. Detecting and Exploiting Algebraic Weaknesses in Cryptosystems II. Algorithms for Placing Modules on a Custom VLSI Chip,* PhD dissertation, Massachusetts Institute of Technology, February 1987.

20. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", *Science*,Vol. 220, No. 4598, May 1983, pp. 671-680.

21. B. W. Kernighan and S. Lin, "An Efficient Heuristic for Partitioning Graphs", *Bell System Technical Journal*,Vol. 49, February 1970, pp. 291-307.

22. C. M. Fudiccia and R. M. Mattheyses, "A Linear-Time Algorithm for Improving Network Partitions", *Proceedings of the 19th Design Automation Conference*, IEEE, 1982, pp. 175-181.

23. T. R. Shiple, "Area Evaluation Metrics for Transistor Placement", Master's thesis, Massachusetts Institute of Technology, June 1987.

24. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Fransisco, California, 1979.

25. D. Hightower, "The Interconnection Problem - A Tutorial", *Proceedings, 10th Design Automation Workshop*, IEEE, 1973, pp. 252-272.

26. C. Y. Lee, "An Algorithm for Path Connections and its Applications", *IRE Transactions on Electronic Computers*,Vol. EC-10, September 1961, pp. 346-365.

27. A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures", *Proceedings, 8th Design Automation Workshop*, IEEE, 1971, pp. 214-224.

28. A. E. Baratz, *Algorithms for Integrated Circuit Signal Routing,* PhD dissertation, Massachusetts Institute of Technology, August 1981.

29. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of*

Computer Algorithms, Addison-Wesley Publishing Company, Reading, Massachusetts, Addison-Wesley Series in Computer Science and Information Processing, Vol. , 1974.

30. R. Rivest, "A "Greedy" Channel Router", *Proceedings of the 19th Design Automation Conference*, IEEE, 1982, pp. 418-424.

31. R. Joobbani and D. Siewiorek, "Weaver: A Knowledge-Based Routing Expert", *Proceedings of the 22nd Design Automation Conference*, IEEE, 1985, pp. 266-272.

32. J. B. Reed, "YACR2: Yet Another Channel Router", Memorandum UCB/ERL M85/16, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, February 1985.

33. R. Chen, Y. Kajitani, and S. Chan, "A Graph-Theoretic Via Minimization Algorithm for Two-Layer Printed Circuit Boards", *IEEE Transactions on Circuits and Systems,*Vol. CAS-30, No. 5, May 1983, pp. 284-299.

34. K. C. Chang and H. C. Du, "A Preprocessor for the Via Minimization Problem", *Proceedings of the 23rd Design Automation Conference*, IEEE, 1986, pp. 702-707.

35. J. Cherry, "CAD Programming in an Object Oriented Programming Environment", *VLSI CAD Tools and Applications*, Kluwer Academic Publishers, Boston, 1986, pp. 265-294.